Projection-based Strictness Analysis

Theoretical and Practical Aspects

Inauguraldissertation zur
Erlangung der Doktorwürde der
Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Ralf Hinze

Bonn
1995

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Berichterstatter: Univ.-Prof. Dr. Armin B. Cremers

2. Berichterstatter: Univ.-Prof. Dr. Robert Giegerich

Tag der mündlichen Prüfung im Hauptfach: 9. November 1995

Tag der mündlichen Prüfung im Nebenfach: 16. November 1995

Für Anja

Abstract

Strictness analysis may be characterized as a compile-time technique for gaining information about lazy functional programs that can be used to improve the generation of code. An instance of strictness analysis is projection-based backward analysis which was motivated in part by the need for a method that could analyse data structures such as lists, trees etc.

The essential flow of information is backward: Given information about the required definedness of the result of a function information about the required definedness of its arguments is determined. Bounds on the definedness of values are specified by projections, a concept borrowed from domain theory. The non-standard semantics of a function, its abstraction, is given by a projection transformer satisfying a certain property of safety.

This work contributes both to the theoretical foundations of backward analysis and to its practical implementation.

A theory of projections is developed collecting and extending results from various sources. Among other things we study the representation of projections and their algebraic properties.

It is shown that every continuous, first-order function possesses a least abstraction which in turn characterizes the function—provided it is strict. Unfortunately, least abstractions fail to be compositional. This desirable property is established by focusing on the stable functions of Berry. Building on the theory of stable functions we then define a non-standard semantics mapping a textual function to its least abstraction.

In order to make the analysis effective infinite domains of projections must be approximated by finite domains. We elaborate in particular on the representation of projections on polymorphic types extending previous work on this subject. An approximation semantics is presented and applied to a variety of functions.

However, due to the exponential growth of the finite domains the analysis proves to be only feasible for small types. As a solution to this problem we propose an alternative technique with a dramatically improved average-case behaviour. The technique is based on a compact representation of abstractions using monotone Boolean functions.

Acknowledgments

I should like to thank my supervisor Prof. Armin B. Cremers for his guidance and encouragement over the last two years of writing this dissertation. Heartfelt thanks are due to Prof. Robert Giegerich for his spontaneous willingness to be co-referee and for directing me to the work of Hughes on the reversal of analysis.

I am particularly grateful to my friends and office mates Martina Doelp, Ulrike Griefahn, Helge Kahler, and Jürgen Kalinski for reading and commenting on parts of the manuscript. Special thanks go to Hermann Stamm-Wilbrandt for many helpful discussions on combinatorial questions.

Discussions with Michael Kettler have been of immense value to me. Thanks for patiently listening to my ideas and for implementing most of them.

Thanks also go to one month old Florian for crying only once in a while. Finally, and most importantly, I wish to thank Anja, for her love and support.

Contents

Chapter 1

Introduction

There are two things which I am confident I can do very well: one is an introduction to any literary work, stating what it is to contain, and how it should be executed in the most perfect manner; the other is a conclusion, shewing from various causes why the execution has not been equal to what the author promised to himself and to the public.

— Samuel Johnson, *Boswell Life vol. 1, p. 291 (1755)*

The history of programming languages is as old as the history of computing devices. The first programming languages were developed as a means to control the behaviour of a particular machine. Naturally, their view of computation reflected intimately the architecture of the underlying hardware. With the development of many different kinds of machines the need for a language freed from the idiosyncrasies of a specific hardware and freeing the programmer from clerical operational details arose.

This need was in part answered by FORTRAN (formula translating system) in the mid-fifties. One of the main features of FORTRAN were expressions introducing a familiar piece of mathematics into the world of computing. The work of translating expressions into instructions of a machine was thus transferred from the individual programmer to the writer of a compiler—as indicated by the name of the language. Although FORTRAN was initially suspected to be inefficient the language became a success in the end.

During the next decades the world saw a proliferation of programming languages. Though different in syntax and in use most of them were still based on the same principles as their common ancestor—the von Neumann computer. Ironically, it was one of the fathers of FORTRAN, John Backus, who criticized in his widely acknowledged Turing Award lecture [15] the conventional style of programming ("[. . .] primitive word-at-a-time programming [. . .]") advocating a more functional style of programming.

The motivation for functional languages corresponds largely to the motivation which led to the development of FORTRAN, namely, to provide a more concise, higher level style of programming resembling traditional mathematical notation. To this end functional languages abandon the usual separation of programs into statements and expressions: Everything is an expression. Expressive power is gained by giving data structures and functions the status of first-class citizens. In a functional language a function which returns a complex data structure, say, a red-black tree is written at ease and by no means unusual.

This allows for a more modular way of programming. A program may be written as a composition of several functions with intermediate data structures serving as clean interfaces between 'adjacent' functions. The evaluation strategy of modern functional programming languages such as MIRANDA[1] [145] or HASKELL [75] further contributes to this style. The output of a function is never produced at once but only on demand and in a piecemeal fashion. For that reason the evaluation strategy is often called 'lazy evaluation'. Lazy evaluation causes the individual functions to be run in strict synchronization closely resembling UNIX[2] pipes.

Since functional languages operate on a much higher level of abstraction than conventional languages they are also much harder to implement efficiently. This comes as no surprise as functional languages have been devised as to concentrate on the task of specifying *what* to compute. The task of working out *how* to compute is largely burdened to the compiler. To put it another way, information which is explicit in a conventional program but only implicit in a functional program must be extracted by means of program analysis. There are many different types of analysis depending on the kind of information one wishes to determine: strictness analysis, termination analysis, update analysis, binding-time analysis to name only a few.

In this dissertation we will be concerned exclusively with strictness analysis. Lazy functional languages free the programmer from considerations about the order of evaluation. Strictness analysis aims at recovering this kind of information.

Plan of the chapter Section 1.1 reviews the parameter passing strategies call-by-value and call-by-need and the notions of strictness and abstract interpretation. Then an ideal-based forward analyser for a simple first-order language with scalar types is developed. Section 1.2 introduces the notions of lazy and eager evaluation and motivates the need for a more general notion of strictness when data structures such as lists or trees come into play. A backward analyser is sketched for a language extended by lists. The section also hints at the main contributions of this work. The first two sections roughly follow the historical development of strictness analysis. A more detailed account of previous work on this subject is given in Section 1.3 which also suggests several dimensions along which different techniques may be classified. Finally, Section 1.4 gives some guidance to this dissertation.

1.1 Forward analysis

Call-by-need vs. call-by-value The family of functional programming languages may be roughly divided into two schools: the pragmatic and the purist school. Members of the former school, for example, STANDARD ML [69] or CAML [151] encourage a functional style of programming but they also contain 'impure' features such as references and exceptions. By contrast pure or if you like purist languages, for example, MIRANDA or HASKELL adhere closely to mathematical traditions the most prominent being the principle of referential transparency. Another distinguishing feature concerns the evaluation strategy: STANDARD ML employs 'eager evaluation' while MIRANDA and HASKELL use 'lazy evaluation'. Loosely speaking a lazy strategy delays the evaluation of expressions until their value is definitely known to be required. For that reason lazy evaluation is sometimes coined demand-driven evaluation.

Let us illustrate the differences between the evaluation strategies by means of a first-order language with scalar types. In this simple case the evaluation strategies eager and lazy boil

[1] MIRANDA is a trademark of Research Software Ltd.
[2] UNIX is a registered trademark of AT&T Bell Laboratories.

down to the parameter passing modes call-by-value and call-by-need.

function call: $\ldots f(e) \ldots$
function definition: $f(x) = \ldots x \ldots$

In call-by-value the formal parameter x is bound to the value of the actual parameter, that is, the parameter is evaluated at the time of the call. Since call-by-value is easy to implement, especially on stock hardware, it is the most common passing mode. In call-by-need the formal parameter is bound to the actual parameter e unevaluated. Only if the formal parameter x is accessed the evaluation of e is initiated, that is, the evaluation takes place at the time of the first access. Call-by-need passing mode is more efficient in the sense that usually fewer reduction steps are required.[3] In the extreme, a function call may terminate under the call-by-need regime while it fails to do so under call-by-value.

Pragmatically speaking lazy evaluation relieves the programmer from considerations about the order of evaluation since an expression is not evaluated until its value is needed. The following definitions which could be part of a numerical package may serve as an example.

```
(&) :: bool bool → bool
a & b = if a then b else False

within :: num num num → bool
within eps a b = b~=0 & abs (a/b-1)<=eps
```

The function `within` checks whether the ratio of `a` and `b` lies near 1. In order to prevent a division by zero the guard `b~=0` was put in front of the test. It is immediate that the definition only works under lazy evaluation.

Lazy evaluation also simplifies program transformation. The rule below is a simple instance of common subexpression elimination.

<u>if</u> e_1 <u>then</u> e_2*e_2 <u>else</u> e_3
$\rightsquigarrow$ <u>let</u> x = e_2 <u>in</u> <u>if</u> e_1 <u>then</u> x*x <u>else</u> e_3

This rule is not valid in an eager language. More precisely the rule is only valid under the proviso that e_2 terminates. But even if we succeed in proving e_2's termination the transformation results in an inefficient program: if e_1 evaluates to false, e_2 was unnecessarily computed.

Call-by-value transformation We have hinted at the advantages of lazy evaluation or call-by-need (cf to the next section for more and probably more convincing arguments). Alas, as ever, the lunch is not quite free. Compared to the cheap call-by-value call-by-need induces a constant overhead at run-time. Call-by-need is usually built on top of call-by-value by introducing unevaluated expressions, called recipes, as first-class objects into the language. Operationally, a recipe is a data structure which holds the information necessary to evaluate the expression in a different context. A recipe typically comprises the bindings of the free variables and a pointer to the compiled code. The primitive operator `freeze` builds a recipe which may be evaluated using `unfreeze`. A lazy program can be transformed into an eager one by freezing the arguments of every function call and unfreezing them upon access.

function call: $\ldots f($`freeze` $e) \ldots$
function definition: $f(x) = \ldots$ `unfreeze` $x \ldots$

[3] In a referentially transparent language the value once computed may be saved in case it is needed again. This is the exact meaning of lazy evaluation or call-by-need. If the value is re-computed each time the variable is accessed one speaks of demand-driven evaluation or call-by-name.

Note that exactly the same technique is employed in environmental machines such as the SECD machine [71] or the Categorical Abstract Machine [37]. Applying the transformation to the above program yields

```
(&) :: bool? bool? → bool
a & b = if unfreeze a then unfreeze b else False

within :: num? num? num? → bool
within eps a b
    = freeze (unfreeze b~=0)
      & freeze (abs (unfreeze a/unfreeze b-1)<=unfreeze eps).
```

By σ? we denote the type of all recipes evaluating to values in σ. Note that the arguments of primitive functions are not delayed since we assume, for the time being, that primitive functions actually require their arguments. The overhead of call-by-need compared to call-by-value is now directly reflected by the appearance of `freeze` and `unfreeze`.

Strictness The technique described is rather extreme: Each argument is delayed whether it is needed or not. The first argument of `&`, for example, is evaluated anyway within the function's body. Hence it is safe to evaluate this argument prior to the function call. In other words, the argument may be passed by value. Consequently, `within` needs its third parameter obviating the respective use of `freeze` and `unfreeze`. These two simple observations already greatly improve the quality of the code.

```
(&) :: bool bool? → bool
a & b = if a then unfreeze b else False

within :: num? num? num → bool
within eps a b
    = b~=0 & freeze (abs (unfreeze a/b-1)<=unfreeze eps)
```

Note that both calling modes are freely mixed within the same program. Peyton Jones and Partain report in [129] that optimizations of this kind result in "solid but modest improvements in execution speed", typically around 20–30%.

The operational notion of neededness corresponds semantically to the notion of strictness. A function $\varphi : D_1 \times \cdots \times D_n \to D$ is called strict in its i-th argument iff

$$(\forall x_1, \ldots, x_n)\; x_i = \bot \Longrightarrow \varphi(x_1, \ldots, x_n) = \bot,$$

that is, the non-termination of the i-th argument implies the non-termination of the entire function call. Strictness is a sufficient (but not necessary) criterion for passing an argument by value. Note that strictness is really an approximation of neededness. The function `f` with `f a = f a` is strict but does not need its parameter. However, the difference between neededness and strictness does not matter as long as we do not distinguish between different types of non-termination.

Abstract interpretation Strictness being a non-trivial property of programs is in general undecidable. Consequently we have to content ourselves with approximations of the actual program behaviour. Strictness analysis in its various forms may be fruitfully regarded as an instance of abstract interpretation which constitutes the formal framework of approximative

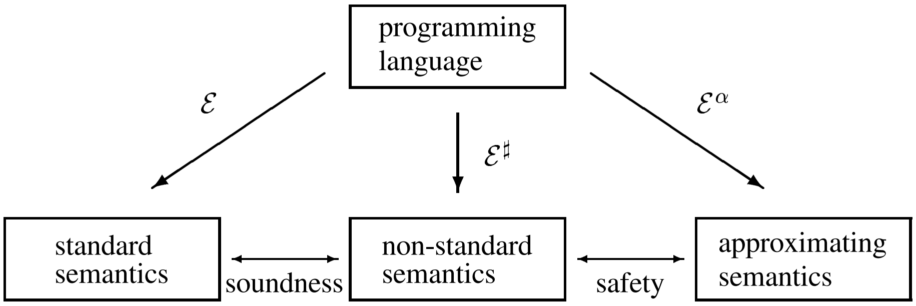


Figure 1.1: Abstract interpretation

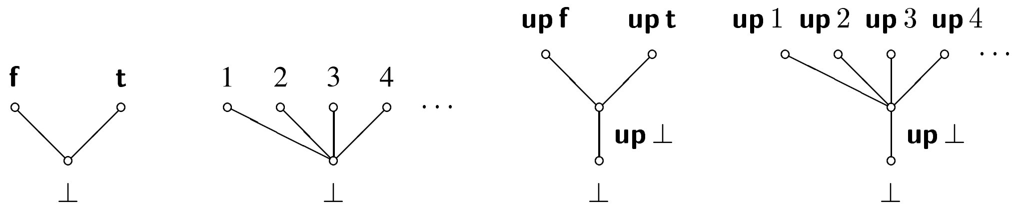
program analysis. Abstract interpretation is a compile-time technique which allows to infer information about a program without executing it. The diagram displayed in Figure 1.1 depicts the ingredients of abstract interpretation. In order to prove an analysis correct we require that both syntax and (standard) semantics of the language are rigidly defined. In this dissertation we use a variant of the Backus-Naur form to specify the syntax of the language and a denotational semantics to specify its meaning.

The properties we are interested in are formalized by means of a non-standard semantics. The proof of soundness demonstrates that the formalization is correct with respect to the standard semantics. Often the non-standard semantics is only a slight variant of the standard one, for example, the standard semantics raised to sets of values (when working in a denotational setting powerdomains take over the rôle of powersets).

The non-standard semantics usually involves infinite domains which have to be approximated in a second step by finite domains, so-called abstract domains. The proof of safety ensures that the approximation is correct with respect to the non-standard semantics. A strictness analyser, for example, is correct if it never falsely signals strictness. It may, of course, fail to report strictness in which case the code generator, which employs the information, misses a chance to optimize. To coin a phrase the safety property ensures that the code generator never goes over the top.

A forward strictness analyser In the sequel we devise a strictness analyser for the first-order language used so far. The material presented is with one exception fairly standard: We base the analysis on the strict language incorporating explicit calls to `freeze` and `unfreeze`. While this choice seems rather unmotivated at the moment it will prove helpful in relating different techniques of analysis, namely the forward analysis presented below and the backward analysis developed in the next section.

Let us briefly sketch the standard and the non-standard semantics. We will not go into detail since we focus on the approximation semantics. The primitive types `bool` and `num` are interpreted by the flat domains $\mathbb{B}_\perp$ and $\mathbb{N}_\perp$ depicted below on the left.



Recipes are interpreted by lifted values. If e denotes the value v, `freeze` e accordingly denotes **up** v. The two diagrams on the right hand side picture the domains $(\mathbb{B}_\perp)_\perp$ and $(\mathbb{N}_\perp)_\perp$ respectively. Since the language has a call-by-value semantics we use smash products and strict continuous functions to model tuples and procedures. Thus the procedure `within` of type `num? num? num?` $\to$ `bool` denotes a function of type $(\mathbb{N}_\perp)_\perp \otimes (\mathbb{N}_\perp)_\perp \otimes (\mathbb{N}_\perp)_\perp \circ\!\!\to \mathbb{B}_\perp$.

As the denotations of procedures are strict functions it is appropriate to rechristen the property of strictness as *lift strictness*. Formally, a function $\varphi : (D_1)_\perp \otimes \cdots \otimes (D_n)_\perp \circ\!\!\to D$ is called lift strict in its i-th argument iff

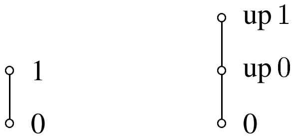
$$(\forall x_1, \ldots, x_n)\; x_i = \perp \Longrightarrow \varphi(\mathbf{up}\, x_1, \ldots, \mathbf{up}\, x_n) = \perp.$$

Turning to the non-standard semantics we note that the above condition may be rewritten using sets of values as

$$\varphi((D_1)_\perp, \ldots, \{\perp\}_\perp, \ldots, (D_n)_\perp) \quad = \quad \{\perp\}.$$

This amounts to raising φ to the Hoare powerdomain which contains as elements Scott-closed subsets or ideals. The fundamental property of ideals is that they are closed 'under going down'. This property will play an important rôle when comparing the relative power of different techniques of analysis.

The definition of lift strictness reveals that we are interested in the definite non-termination of expressions.[4] Thus we use the following abstract domains.



The order on the left abstracts flat domains like the domain of truth values $\mathbb{B}_\perp$. The interpretation of 0 is 'definitely does not terminate' while 1 reads as 'may or may not terminate'. The order on the right models unevaluated expressions over base types. The value up 0 represents recipes which definitely do not terminate if forced. Accordingly, up 1 represents recipes which may or may not terminate if forced. The value 0, which plays little rôle in practice, models the situation when the construction of the recipe fails, for example, due to lack of memory.

We can determine the lift strictness of a function by testing its abstraction. If φ has the type $(\mathbb{N}_\perp)_\perp \otimes (\mathbb{N}_\perp)_\perp \circ\!\!\to \mathbb{N}_\perp$ then its forward abstraction, say, φ^{f} is of type $\mathbf{2}_\perp \otimes \mathbf{2}_\perp \circ\!\!\to \mathbf{2}$.

$$\begin{aligned}
\varphi^{\mathrm{f}}\,(\mathrm{up}\,0)\,(\mathrm{up}\,1) = 0 \quad &\Longrightarrow \quad \varphi \text{ is lift strict in its 1. argument} \\
\varphi^{\mathrm{f}}\,(\mathrm{up}\,1)\,(\mathrm{up}\,0) = 0 \quad &\Longrightarrow \quad \varphi \text{ is lift strict in its 2. argument}
\end{aligned}$$

Note that both $\varphi^{\mathrm{f}}\,0\,a$ and $\varphi^{\mathrm{f}}\,a\,0$ trivially evaluate to 0 since the language has a call-by-value semantics. The remaining cases are interesting in their own right.

$$\begin{aligned}
\varphi^{\mathrm{f}}\,(\mathrm{up}\,0)\,(\mathrm{up}\,0) = 0 \quad &\Longrightarrow \quad \varphi \text{ is joint strict} \\
\varphi^{\mathrm{f}}\,(\mathrm{up}\,1)\,(\mathrm{up}\,1) = 0 \quad &\Longrightarrow \quad \varphi \text{ is divergent}
\end{aligned}$$

A function is joint strict in two or more arguments iff at least one of these arguments is needed. The conditional, for example, is joint strict in the second and third argument. A function is called divergent iff it is undefined irrespective of its argument.

Let us turn to the problem of determining the abstraction of a procedure by just inspecting the program text. We will see that the definition of strictness analysis is very much akin to the

[4] This is in contrast to totality analysis where one is interested in definite termination.

definition of the standard semantics. The approximation semantics enjoys, for example, the property of compositionality, that is, the meaning of an expression is determined by the meaning of its immediate subexpressions. The flow of control is essentially forward: Knowing the abstract values of the arguments the abstract value of the result is computed. For that reason the analysis is often classified as *forward*. [Surprisingly, the same information may also be obtained backwards (cf to the next section).] Pictorially, strictness information is propagated from the leaves of an abstract syntax tree to its root. Consider, for example, the definition of a lazy multiplication.

```
times :: num num → num
times x y = if x=0 then 0 else x*y
```

Explicating the `unfreeze` operators the program becomes

```
times :: num? num? → num
times x y = if unfreeze x=0 then 0 else unfreeze x*unfreeze y.
```

It is not hard to see that `times` is lift strict in its first argument and not lift strict in its second. The analysis of the first property is depicted in Figure 1.2.

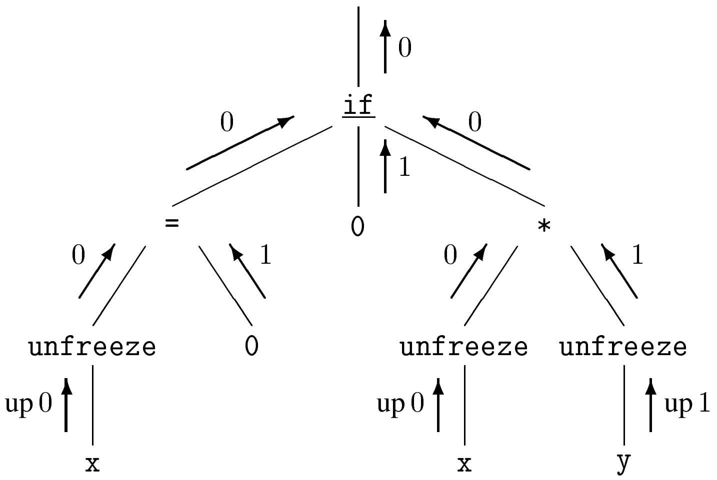


Figure 1.2: Forward analysis of `times`

In order to simplify the definition of the approximation semantics we assume that the formal parameters of a procedure are numbered from left to right. Let a be a vector $a_1 \ldots a_n$ on n abstract values, then $\mathcal{E}^{\mathrm{f}}[\![e]\!]\, a$ denotes the abstract value of e with respect to a. Note that $\mathcal{E}^{\mathrm{f}}[\![e]\!]\, a_1 \ldots 0 \ldots a_n$ trivially evaluates to 0. If e is a variable we have,

$$\mathcal{E}^{\mathrm{f}}[\![x_i]\!]\, a \quad = \quad a_i.$$

A constant always terminates, hence its abstract value is 1 independent of a.

$$\mathcal{E}^{\mathrm{f}}[\![c]\!]\, a \quad = \quad 1$$

Primitive operations such as = or + are defined only if both arguments are defined, so

$$\mathcal{E}^{\mathrm{f}}[\![e_1 \texttt{ + } e_2]\!]\, a \quad = \quad \mathcal{E}^{\mathrm{f}}[\![e_1]\!]\, a \sqcap \mathcal{E}^{\mathrm{f}}[\![e_2]\!]\, a.$$

The `freeze` operator is particularly easy to model since the abstract domain already includes its abstraction.

$$\mathcal{E}^{\mathrm{f}}[\![\texttt{freeze}\ e]\!]\, a \;=\; \mathrm{up}(\mathcal{E}^{\mathrm{f}}[\![e]\!]\, a)$$

The operator `unfreeze` constitutes the counterpart of `freeze`. Accordingly, its abstraction is christened 'down'.

$$\begin{aligned}\mathrm{down}(0) &= 0\\ \mathrm{down}(\mathrm{up}\, a) &= a\end{aligned}$$

Using 'down' we can write the abstract interpretation of `unfreeze` as follows.

$$\mathcal{E}^{\mathrm{f}}[\![\texttt{unfreeze}\ e]\!]\, a \;=\; \mathrm{down}(\mathcal{E}^{\mathrm{f}}[\![e]\!]\, a)$$

Finally, the conditional $\underline{\texttt{if}} \cdot \underline{\texttt{then}} \cdot \underline{\texttt{else}}\cdot$ fails to terminate if either the condition does not terminate or both branches do not terminate. This amounts to saying that the conditional is strict in the first argument and joint strict in its second and third. It is helpful to introduce an auxiliary notation.

$$\begin{aligned}0 \longrightarrow a &= 0\\ 1 \longrightarrow a &= a\end{aligned}$$

The abstraction of the conditional is then defined by

$$\mathcal{E}^{\mathrm{f}}[\![\underline{\texttt{if}}\ e_1\ \underline{\texttt{then}}\ e_2\ \underline{\texttt{else}}\ e_3]\!]\, a \;=\; \mathcal{E}^{\mathrm{f}}[\![e_1]\!]\, a \longrightarrow (\mathcal{E}^{\mathrm{f}}[\![e_2]\!]\, a \sqcup \mathcal{E}^{\mathrm{f}}[\![e_3]\!]\, a).$$

Let us apply the analysis to the procedure `times` defined above. If e is the body of the function `times`, then

$$\begin{aligned}\texttt{times}^{\mathrm{f}}\, a_1\, a_2 &= \mathcal{E}^{\mathrm{f}}[\![e]\!]\, a_1\, a_2\\ &= \mathrm{down}\, a_1 \sqcap 1 \longrightarrow 1 \sqcup (\mathrm{down}\, a_1 \sqcap \mathrm{down}\, a_2)\\ &= \mathrm{down}\, a_1.\end{aligned}$$

Clearly, $\texttt{times}^{\mathrm{f}}\,(\mathrm{up}\,0)\,(\mathrm{up}\,1) \;=\; 0$ and $\texttt{times}^{\mathrm{f}}\,(\mathrm{up}\,1)\,(\mathrm{up}\,0) \;=\; 1$, so we have inferred that `times` is definitely lift strict in its first and probably not lift strict in its second argument.

Dealing with recursion The attentive reader may have noticed that we kept quiet about the analysis of recursive functions. As to be expected, recursive functions give rise to recursive abstract functions. Consider, for example,

```
countdown :: num num → num
countdown x y = if x=0 then y else countdown (x-1) y,
```

whose call-by-value variant is[5]

```
countdown :: num? num? → num
countdown x y = if unfreeze x=0 then unfreeze y
                else countdown (freeze (unfreeze x-1)) y.
```

[5]The example involves a simple improvement: The expression `freeze (unfreeze` x`)` is simplified to x.

Applying the analysis to the definition of `countdown` yields

$$\begin{aligned}
&\texttt{countdown}^{\mathrm{f}}\, a_1\, a_2\\
=\;& \mathrm{down}\, a_1 \sqcap 1 \longrightarrow \mathrm{down}\, a_2 \sqcup \texttt{countdown}^{\mathrm{f}}\, (\mathrm{up}(\mathrm{down}\, a_1 \sqcap 1))\, a_2\\
=\;& \mathrm{down}\, a_1 \longrightarrow \mathrm{down}\, a_2 \sqcup \texttt{countdown}^{\mathrm{f}}\, a_1\, a_2.
\end{aligned}$$

As the definition of $\texttt{countdown}^{\mathrm{f}}$ only involves monotonic functions and since we are working on finite domains the least solution of the recursive equation may be calculated using a simple fixpoint iteration. The successive approximations to $\texttt{countdown}^{\mathrm{f}}$ are tabulated below.

a_1	0	$\mathrm{up}\,0$	$\mathrm{up}\,0$	$\mathrm{up}\,1$	$\mathrm{up}\,1$
a_2	0	$\mathrm{up}\,0$	$\mathrm{up}\,1$	$\mathrm{up}\,0$	$\mathrm{up}\,1$
$\texttt{countdown}^{\mathrm{f}}_1\, a_1\, a_2$	0	0	0	0	0
$\texttt{countdown}^{\mathrm{f}}_2\, a_1\, a_2$	0	0	0	0	1
$\texttt{countdown}^{\mathrm{f}}_3\, a_1\, a_2$	0	0	0	0	1

Since the second approximation is identical to the third the least fixpoint of the recursive function is reached in one step.

Let us conclude the section with some remarks about the complexity of forward analysis. The tabulation used above has the obvious disadvantage that comparing two successive approximations is exponential in the number of arguments to the functions. Though more clever representations have been devised it can be shown that the problem of checking the equality of two two-valued functions of n arguments if NP-complete [77].

Generally, we do not need to know the complete fixpoint. With respect to the generation of code joint strictness is of no interest, so we may confine ourselves to evaluating calls of the form $\varphi^{\mathrm{f}}\,(\mathrm{up}\,1)\ldots(\mathrm{up}\,0)\ldots(\mathrm{up}\,1)$. Unfortunately, the evaluation of a monotone Boolean function at any point is DEXPTIME-complete in the length of the function definition [76].

Nonetheless, it is generally assumed that strictness analysis is tractable (at least in the first-order case), since user-defined functions are unlikely to exhibit such a worst-case behaviour. Note, that an integral part of most functional language compilers, the Milner-style type-inference algorithm, was recently shown [103] to be DEXPTIME-complete, as well.

1.2 Backward analysis

Eager vs. lazy evaluation Things become considerably more interesting if we extend the first-order language by structured values such as lists or trees. Lists, for example, are constructed using the empty list `[]` and the operator '`:`' (pronounce 'cons') which puts an element in front of a list. Lists are analysed by means of a <u>`case`</u>-expression.

```
list construction:  [], e1 : e2
list analysis:      case e of []  → ...  |
                              a:x → ... a ... x ...
```

The discussion of evaluation strategies applies in particular to the new language constructs. As before there are in principle two alternatives: the arguments of the '`:`' operator are evaluated at the time of the call (eager evaluation) or at the first time of access, that is, in the second branch of a <u>`case`</u>-expression (lazy evaluation).

The strength of lazy evaluation becomes especially apparent in connection with structured values. Though we do not intend to enter a discussion about the virtues of non-strict languages (cf to [83] for a lucid account), let us again briefly hint at some advantages. Broadly speaking lazy evaluation allows to structure programs in a more modular way thereby supporting the re-use of software. Consider as simple a task as generating the list of numbers in increasing order from m to n inclusive, traditionally written as $[m\,..\,n]$. Instead of defining one monolithic function we may decompose the task by programming a function which generates the list of numbers from m and a second which takes an initial segment of a given length.

```
from :: num → [num]
from n = n:from (n+1)

take :: num [α] → [α]
take n x = if n=0 then []
           else case x of [] → [] |
                              a:w → a:take (n-1) w
```

The list $[m\,..\,n]$ is then defined by `take` $(n$-m+1) (`from` m). This scheme applies to a number of problems: generating the list of all primes and taking an initial segment of length n, generating a game tree and pruning it to a given depth etc. Lazy evaluation allows to separate issues of logic and control. The logic component specifies *what* elements a data structure contains while the control specifies *how* a data structure is evaluated. It is immediate that a 'control function' such as `take` may be applied to a variety of situations. Conversely, the control may also be easily exchanged: The list of primes could be evaluated until a given bound is exceeded, a game tree could be traversed depending on the static evaluation of positions etc.

Call-by-value transformation The transformation introduced in the last section extends easily to the new language constructs. We freeze the arguments of the constructor '`:`' and unfreeze them upon access in the second branch of a case-expression.

list construction:	`[]`, `freeze` e_1`:freeze` e_2
list analysis:	case e of `[]` → ... |
	a:x → ... `unfreeze` a ... `unfreeze` x ...

Applying the transformation to `from` and `take` considerably blows up their definitions. It should be noted, however, that the blow-up reflects the overhead of lazy evaluation compared to eager evaluation. [We tacitly ignore that the definition of `from` does not make sense in a strict language.]

```
from n = n:freeze (from (freeze (unfreeze n+1)))

take n x = if unfreeze n=0 then []
           else case unfreeze x of
                [] → [] |
                a:w → a:freeze (take (freeze (unfreeze n-1)) w)
```

Strictness is context sensitive The definition of strictness respectively of lift strictness is, of course, also applicable to functions working on structured values. List concatenation serves as an example for a function having both strict and non-strict arguments.

```
append :: [α] [α] → [α]
append x y = case x of [] → y |
                       a:w → a:append w y
```

Because the first parameter is subject to case analysis, append is strict in its first argument. Strictness in the second argument does not hold, since append [1] $\perp = 1{:}\perp \neq \perp$.

Strictness asks what happens if an argument is totally undefined. When data structures come into play we may differentiate between various degrees of definedness. Let us consider some examples.

```
sum :: [num] → num
sum x = case x of [] → 0 |
                  a:w → a+sum w

length :: [α] → num
length x = case x of [] → 0 |
                     a:w → 1+length w
```

The function sum requires its argument to be totally defined while length only demands that the tail of the argument list is defined (recursively). Functions of the latter kind are termed *tail strict*. Accordingly a function is called *head strict* if the head of the list must be proper (recursively). It is not hard to see that sum combines both of these properties. Typical examples for exclusively head strict functions are filters where the head of the list is involved in a conditional expression.

```
until0 :: [num] → [num]
until0 x = case x of [] → [] |
                     a:w → if a=0 then [] else a:until0 w

and :: [bool] → bool
and x = case x of [] → True |
                  a:w → a & and w
```

Returning to the concatenation of lists we must state that append exhibits neither of these properties. Things become different if we take the context in which the call of append is situated into account. If append is embedded in a tail strict context, say, length (append e_1 e_2), list concatenation becomes strict and tail strict in both arguments. A head strict context like until0 (append e_1 e_2) implies head strictness of both arguments. Note that even the strictness of the second parameter depends crucially on the context. The behaviour of append may be summarized as follows ($\mathtt{append}^\flat$ denotes the backwards abstraction of append).

$$
\begin{aligned}
\mathtt{append}^\flat(\mathrm{HT}) &= \mathrm{HT!}\ \mathrm{HT!}\\
\mathtt{append}^\flat(\mathrm{H}) &= \mathrm{H!}\ \mathrm{H?}\\
\mathtt{append}^\flat(\mathrm{T}) &= \mathrm{T!}\ \mathrm{T!}\\
\mathtt{append}^\flat(\mathrm{L}) &= \mathrm{L!}\ \mathrm{L?}
\end{aligned}
$$

The letters H, T, HT, and L stand for head strictness, tail strictness, combined head and tail strictness and no strictness (L like list). Call-by-value passing mode (or strict lift) is indicated by an exclamation mark, call-by-need (or lazy lift) by a question mark, respectively.[6]

[6] The term 'call-by-value' is probably misleading when complex data structures are involved. Passing the first argument to append by value does *not* mean, that the entire list is evaluated in advance; the argument is merely reduced to *head normalform*. An expression of list type is in head normalform if it is either the empty list [] or else of the form $e_1 : e_2$, which is necessary and sufficient for the case analysis to decide which alternative applies. The evaluation of the components of the list is only specified by H, T, HT, and L.

Thus if we compile a call to `append` in a tail strict context, we may pass both arguments by value and additionally we may evaluate the spine of both lists in advance. Especially the last optimization is promising since the savings are proportional to the length of the list.

The function `append` propagates the demand on the result to both of its arguments. The function `take` is less 'well-behaved'. It does not preserve tail strictness, since the result is generally a true prefix of the argument.

$$
\begin{aligned}
\texttt{take}^\flat(\mathrm{HT}) &= \texttt{num!}\ \mathrm{H?} \\
\texttt{take}^\flat(\mathrm{H}) &= \texttt{num!}\ \mathrm{H?} \\
\texttt{take}^\flat(\mathrm{T}) &= \texttt{num!}\ \mathrm{L?} \\
\texttt{take}^\flat(\mathrm{L}) &= \texttt{num!}\ \mathrm{L?}
\end{aligned}
$$

Let us briefly discuss the use of strictness information in the generation of code. Since a function behaves differently in different contexts it stands to reason to generate specialized code for each context in which it is called. Consider the expression

```
sum (freeze (take (freeze n) (freeze (from (freeze 1))))),
```

which computes $1 + \cdots + n$. Now, as `sum` is head and tail strict the argument of `take` can be evaluated head strict which in turn implies that `from` definitely requires its argument. Building on these observations we may optimize the above expression to

```
sum (takeHT n (freeze (fromH 1))),
```

where `sum`, `takeHT`, and `fromH` are defined as follows.

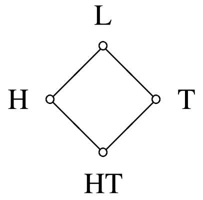
```
fromH n = n:freeze (from (n+1))

takeHT n x = if n=0 then []
             else case unfreeze x of [] → [] |
                                     a:w → a:takeHT (n-1) w

sum x = case x of [] → 0 |
                  a:w → a+sum w
```

The savings in terms of `freeze` and `unfreeze` are impressing: The original expression executes approximately $3n$ `freeze` and $6n$ `unfreeze` operations whereas the optimized variant calls both operations n times only. There is, however, a slight slip in the argument. The original expression runs in $O(1)$ space while the improved one requires $O(n)$ space since the list `[1..`n`]` is constructed in its entirety before `sum` is called. The advantage of lazy evaluation, namely the interleaving of the generation and the consumption of data structures is compromised by the above optimization. To avoid this problem we could decide to delay always the second, that is, the recursive argument of the '`:`' operation.[7] In other words, we consider only head strictness and ignore tail strictness.

[7] This heuristic applies to arbitrary data types. By delaying each of the recursive components of a data type we can guarantee that the space complexity increases by at most a constant.

A backward analyser Let us turn to the automatic inference of context dependent strictness properties. We will be largely informal leaving the rigorous treatment to the chapters to follow. Similar to forward analysis we work on finite abstract domains. The abstract domain of lists, for example, contains the contexts H, T, HT, and L. The ordering of the contexts becomes apparent if we take a look at their definitions. For the time being it suffices to regard them as type expressions incorporating strictness annotations.

$$\begin{aligned} \mathrm{HT} &= \mu\beta.\texttt{[]} \mid \texttt{num!}:\beta! \\ \mathrm{T} &= \mu\beta.\texttt{[]} \mid \texttt{num?}:\beta! \\ \mathrm{H} &= \mu\beta.\texttt{[]} \mid \texttt{num!}:\beta? \\ \mathrm{L} &= \mu\beta.\texttt{[]} \mid \texttt{num?}:\beta? \end{aligned}$$



Since call-by-value is weaker compared to call-by-need, we put $! \sqsubseteq ?$ obtaining the lattice at the right.

The examples of $\texttt{append}^\flat$ and $\texttt{take}^\flat$ show that abstractions revert domain and codomain of the original function: A context for the result is mapped to contexts for the arguments. Consequently, the natural order of analysis is backward from the root of an expression to the leaves. The principle of backward analysis is best introduced by means of an example. Consider the `reverse` function.

```
reverse :: [α] → [α]
reverse x = case x of [] → [] |
                      a:w → append (reverse w) [a]
```

Since `reverse` is recursively defined, calculation of its abstraction involves a fixpoint process. Let us assume that the last step in the iteration has yielded the following result.

$$\begin{aligned} \texttt{reverse}^\flat(\mathrm{HT}) &= \mathrm{HT}! \\ \texttt{reverse}^\flat(\mathrm{T}) &= \mathrm{T}! \\ \texttt{reverse}^\flat(\mathrm{H}) &= \mathrm{T}! \\ \texttt{reverse}^\flat(\mathrm{L}) &= \mathrm{T}! \end{aligned}$$

The analysis of `reverse` with respect to the contexts H and T is depicted in Figure 1.3. Note that the analysis is based as usual on the call-by-value variant of the program. The <u>`case`</u> construct which forms the root of `reverse`'s function body propagates its context to each of the branches. As the first branch is rather uninteresting we focus on the second one. Knowing the abstractions of `append` and `reverse` we are able to push the root context towards the leaves. If a `freeze` node is traversed the passing mode of the current context, '!' or '?', is simply stripped off, that is, we consciously ignore (for the time being) whether the subexpression is actually required or not. The analysis of the subexpression is based on the assumption that it is definitely needed. Note that the contexts H, T, HT, and L encode the behaviour of '`:`', so there is no need to derive an abstraction for '`:`'.

Regardless of what the term 'backward analysis' may suggest this is only half of the work. On the way back to the root we have to combine the results of the different branches. An (intermediate) result is an environment mapping the free variables to contexts. If a variable appears more than once in an expression the presumably different demands must be combined according to the flow of control. We postpone a detailed discussion of the necessary operations since the function body of `reverse` contains every variable exactly once. The combination of the abstract environments on the way back to the root is depicted in Figure 1.4. At this

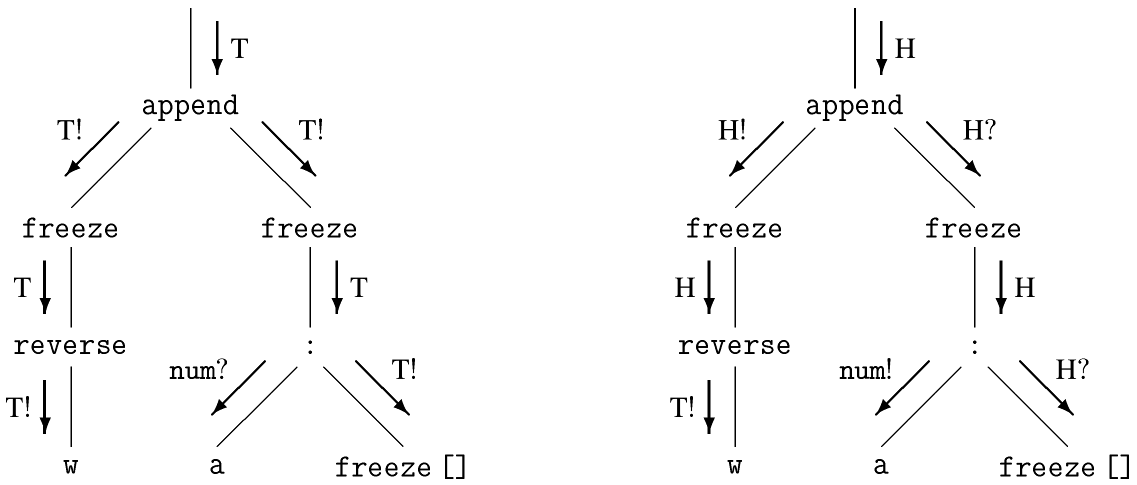


Figure 1.3: Backward analysis of `reverse` (propagating the root context to the leaves)

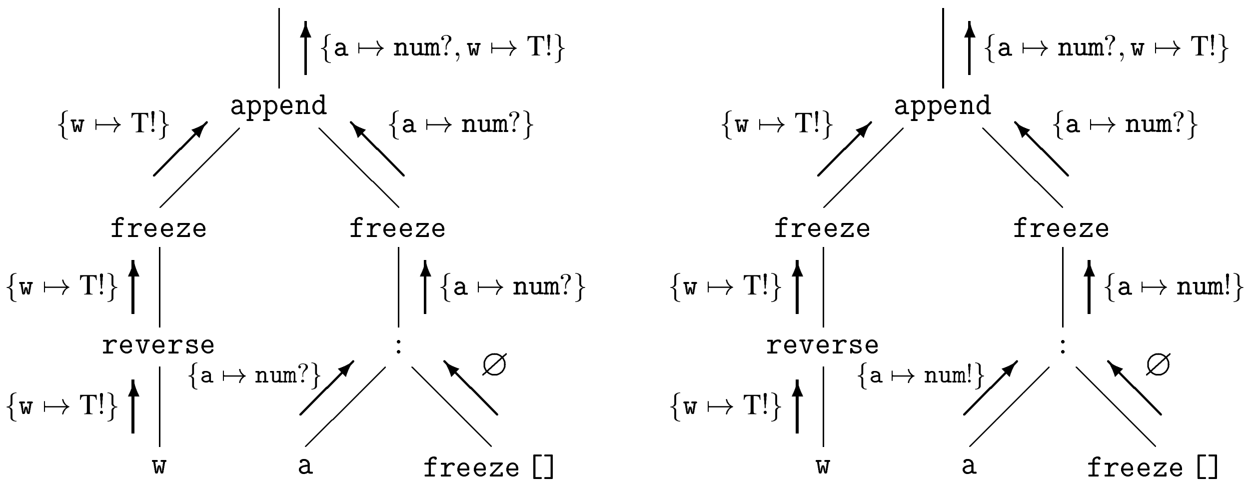


Figure 1.4: Backward analysis of `reverse` (combining the abstract environments)

point the passing modes '!' and '?' get involved. In the first pass we have calculated the contexts for the leaves on the assumption that each subexpression is really evaluated. If this is not the case, that is, the passing mode equals '?', we have to adjust the results. This can be done by setting the passing mode of every free variable to call-by-need, for example, $\{\mathtt{a} \mapsto \mathtt{num}!\}$ is turned to $\{\mathtt{a} \mapsto \mathtt{num}?\}$. This operation is commonly called the *guard operation*.

To determine the context for the argument of `reverse`, one has to combine the contexts for '`a`' and '`w`' which results in the same abstraction we have started with. Alas, we have found a fixpoint.

The attentive reader may have noticed that we dealt with the approximation semantics first. An abstract value, that is, a context may be interpreted by a projection, a concept borrowed from domain theory. A projection is a reductive[8], idempotent, continuous function capturing the operational notion of evaluation. Consequently, the non-standard semantics of a function, say, φ is a projection transformer τ satisfying the following safety condition.

$$(\forall\pi)\ \pi \circ \varphi = \pi \circ \varphi \circ \tau(\pi)$$

The transformer τ maps a demand on the result of φ to a demand which may be applied to the argument without altering the meaning of the function call. Since we restrict ourselves to strict functions the argument may, in fact, be evaluated prior to the call. Note that τ is not determined by the above equation: A projection transformer which trivially satisfies the safety condition is $\tau(\pi) = \textbf{id}$. The question naturally arises as to whether there is a least transformer satisfying the safety condition. The answer is in the affirmative: We will prove in Chapter 5 that every first-order function possesses a least abstraction. Furthermore the least abstraction determines in turn the function implying the equivalence of the standard and the non-standard semantics.

Generic backward analysis The above examples show that backward strictness analysis is able to determine many interesting strictness properties. A major drawback of the technique is its computational complexity: Let n be the size of the function's result type. Since the domain of the context transformer grows exponentially with n, the method is only feasible for 'small' types like lists or trees. As an extreme example consider a parser for the language used in this dissertation which turns a sequence of characters into an abstract syntax tree. A suitable data type for the abstract syntax gives rise to 13.731.766.272 different contexts (cf Section 7.6.6).

R. Kubiak *et. al.* [105] suggest to use minimal function graphs for finding fixpoints. While this method allows to focus on selected arguments thereby avoiding the computation of the whole fixpoint, it is not of much help in connection with separate compilation where strictness information has to be passed from one module to another.

As one of the main contributions of this work we present a variant of backward strictness analysis, termed *generic analysis*, which exhibits a dramatically improved average-case behaviour. The speedup is gained by employing a compact representation of abstractions. First note that abstractions like $\mathtt{append}^\flat$ and $\mathtt{reverse}^\flat$ are monotonic. Furthermore note that list contexts differ only in their passing modes. Thus it is possible to represent an abstraction by a collection of monotonic, Boolean valued functions.

$$\begin{aligned}
\mathtt{append}^\flat(\mu\beta.\,\mathtt{[]} \mid \mathtt{num}a : \beta b) &= (\mu\beta.\,\mathtt{[]} \mid \mathtt{num}a : \beta b)!\,(\mu\beta.\,\mathtt{[]} \mid \mathtt{num}a : \beta b)b \\
\mathtt{reverse}^\flat(\mu\beta.\,\mathtt{[]} \mid \mathtt{num}a : \beta b) &= (\mu\beta.\,\mathtt{[]} \mid \mathtt{num}(a \vee b) : \beta!)!
\end{aligned}$$

[8] A function, say, φ is reductive iff $(\forall x)\ \varphi(x) \sqsubseteq x$.

Instantiating the Boolean variables a and b with '!' and '?' yields the original definitions of append$^\flat$ and reverse$^\flat$. Generic backward strictness analysis generalizes the original method to parameterized contexts containing Boolean terms instead of concrete passing modes. This procedure is best illustrated by means of an example. Let $\Phi(a, b) = \mu\beta.\,\texttt{[]} \mid \texttt{num}a\!:\!\beta b$. The propagation of the outermost context to the leaves works in the same manner as before (cf Figure 1.5). On the way back to the root the passing modes must be incorporated into the

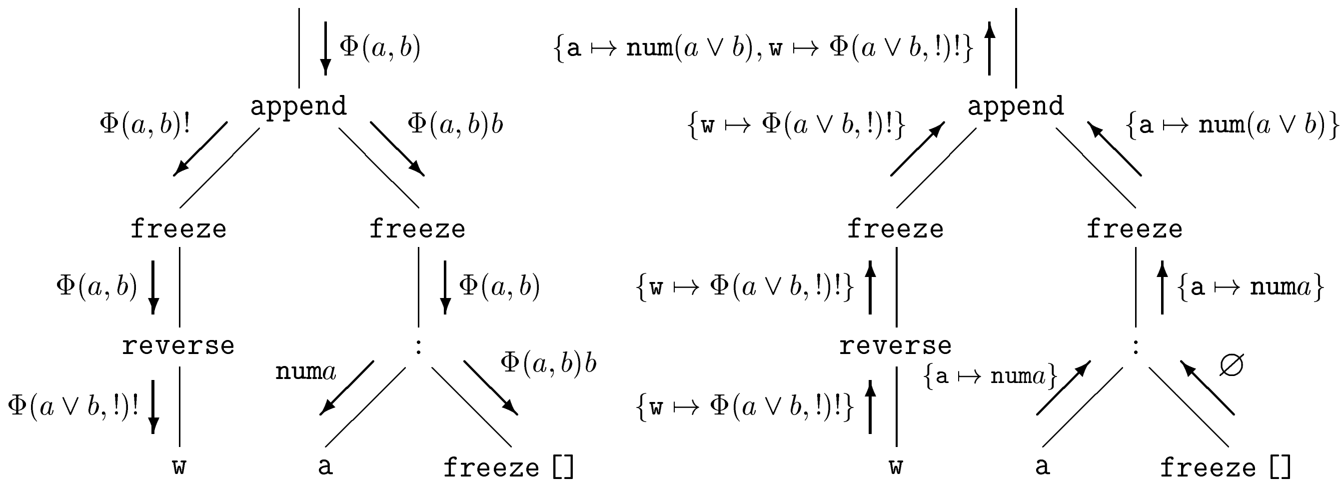
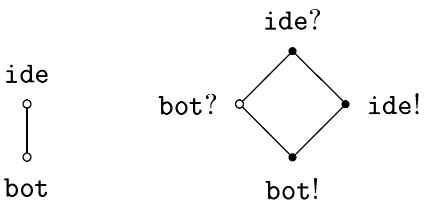


Figure 1.5: Generic backward analysis of reverse

abstract environments. An analysis of the guard operation shows that this operation corresponds to a simple disjunction. All in all we obtain the same result for reverse as the ordinary analysis using only a single pass through the function body.

Relating forward and backward analysis We have introduced two different analyses. It remains to compare the relative power of forward, ideal-based and backward, projection-based analysis. In the sequel we formally develop a backward analyser for the restricted first-order language used in the last section which is then compared to the forward analyser. The following abstract domains are used.



The order on the left abstracts flat domains. The context ide corresponds to 'no evaluation' while bot represents a non-terminating context. Combining ide and bot with the passing modes '!' and '?' yields the order on the right which abstracts unevaluated expressions over base types. Note that we will only use the shaded elements in the sequel. Let $\varphi^\flat$ be the backward abstraction of $\varphi : (\mathbb{N}_\bot)_\bot \otimes (\mathbb{N}_\bot)_\bot \multimap \mathbb{N}_\bot$, then we may determine the lift strictness of

φ as follows.

$$\begin{array}{lcl} \varphi^\flat\ \mathtt{ide} = \mathtt{ide!}\,\mathtt{ide!} & \Longrightarrow & \varphi \text{ is lift strict in both arguments} \\ \varphi^\flat\ \mathtt{ide} = \mathtt{ide!}\,\mathtt{ide?} & \Longrightarrow & \varphi \text{ is lift strict in its 1. argument} \\ \varphi^\flat\ \mathtt{ide} = \mathtt{ide?}\,\mathtt{ide!} & \Longrightarrow & \varphi \text{ is lift strict in its 2. argument} \\ \varphi^\flat\ \mathtt{ide} = \mathtt{ide?}\,\mathtt{ide?} & \Longrightarrow & \varphi \text{ is neither lift strict in the 1. nor in the 2. argument} \end{array}$$

[Note that $\varphi^\flat$ `bot` trivially evaluates to `bot! bot!`.] Contrary to forward analysis the strictness properties are obtained via a *single* call to the abstraction. However, upon inspection we find that the property of joint strictness is missing in the list above. It turns out that joint strictness cannot be represented by a context of the form $\kappa_1\,\kappa_2$. Instead we must use the disjunction $\mathtt{ide!}\,\mathtt{ide?} \sqcup \mathtt{ide?}\,\mathtt{ide!}$ which may be interpreted as 'either the first or the second component is required but we do not know which one'. Generally, the abstraction of a tuple may only be represented by a *set of* tuples of abstractions.

$$\begin{array}{lcl} \varphi^\flat\ \mathtt{ide} = \mathtt{bot!}\,\mathtt{bot!} & \Longrightarrow & \varphi \text{ is divergent} \\ \varphi^\flat\ \mathtt{ide} = \mathtt{ide!}\,\mathtt{ide?} \sqcup \mathtt{ide?}\,\mathtt{ide!} & \Longrightarrow & \varphi \text{ is joint strict} \end{array}$$

Turning to the analysis of a textual function we assume as in the preceding section that the formal parameters of a procedure are numbered from left to right. The backward abstraction of the expression e is denoted by $\mathcal{E}^\flat[\![e]\!]$. Note that $\mathcal{E}^\flat[\![e]\!]$ `bot` trivially evaluates to $\mathtt{bot!}\ldots\mathtt{bot!}$. If e is a variable we have

$$\mathcal{E}^\flat[\![x_i]\!]\,\kappa \;=\; \mathtt{ide?}\ldots\kappa\ldots\mathtt{ide?},$$

that is, the i-th argument is required while the remaining ones are not needed. A constant imposes no demand on the argument tuple.

$$\mathcal{E}^\flat[\![c]\!]\,\kappa \;=\; \mathtt{ide?}\ldots\mathtt{ide?}$$

To give the abstract interpretation of a strict primitive function we need a new operator, $\&$, which combines two contexts 'conjunctively'. For the moment it is safe to identify $\&$ with the meet operator $\sqcap$.[9]

$$\mathcal{E}^\flat[\![e_1\ \mathtt{+}\ e_2]\!]\,\kappa \;=\; \mathcal{E}^\flat[\![e_1]\!]\,\kappa \,\&\, \mathcal{E}^\flat[\![e_2]\!]\,\kappa$$

A demand on an unevaluated expression is either equal to $\kappa!$ or to $\kappa?$. Consequently, we distinguish two cases: If we know that `freeze` e is definitely evaluated we calculate the demand induced by e, otherwise we know that none of the arguments is required.

$$\begin{array}{lcl} \mathcal{E}^{\mathrm{f}}[\![\mathtt{freeze}\ e]\!]\,(\kappa!) & = & \mathcal{E}^\flat[\![e]\!]\,\kappa \\ \mathcal{E}^{\mathrm{f}}[\![\mathtt{freeze}\ e]\!]\,(\kappa?) & = & \mathtt{ide?}\ldots\mathtt{ide?} \end{array}$$

The operator `unfreeze` triggers the evaluation of a recipe. Hence we define

$$\mathcal{E}^\flat[\![\mathtt{unfreeze}\ e]\!]\,\kappa \;=\; \mathcal{E}^\flat[\![e]\!]\,(\kappa!).$$

The conditional is really an instance of the <u>`case`</u>-expression, so we may profit from the analysis of `reverse` discussed above. The context on the result is first propagated to the two branches. Since we do not know in advance which branch is executed the computed demands

[9] The identification of $\&$ and $\sqcap$ is safe as long as we restrict ourselves to so-called smash projections which correspond to ideals used in the forward analysis.

are combined 'disjunctively' using the join operator $\sqcup$. However, we know that the condition *and* one of the branches is evaluated which can be expressed as follows

$$\mathcal{E}^\flat[\![\underline{\mathtt{if}}\ e_1\ \underline{\mathtt{then}}\ e_2\ \underline{\mathtt{else}}\ e_3]\!]\ \kappa \quad = \quad \mathcal{E}^\flat[\![e_1]\!]\ \mathtt{ide}\ \&\ (\mathcal{E}^\flat[\![e_2]\!]\ \kappa \sqcup \mathcal{E}^\flat[\![e_3]\!]\ \kappa).$$

This completes the presentation of the backward analyser. Let us apply the analyser to the procedure `times` defined in the preceding section to see whether we obtain as strong a result as in the forward case.

$$\begin{aligned} \mathtt{times}^\flat\ \mathtt{ide} &= \mathcal{E}^\flat[\![e]\!]\ \mathtt{ide} \\ &= (\mathtt{ide!}\ \mathtt{ide?}\ \&\ \mathtt{ide?}\ \mathtt{ide?})\ \&\ (\mathtt{ide?}\ \mathtt{ide?} \sqcup \mathtt{ide!}\ \mathtt{ide?}\ \&\ \mathtt{ide?}\ \mathtt{ide!}) \\ &= \mathtt{ide!}\ \mathtt{ide?} \end{aligned}$$

It follows immediately that `times` is definitely lift strict in its first and probably not lift strict in its second argument. This is exactly the same result as inferred by the forward analyser.

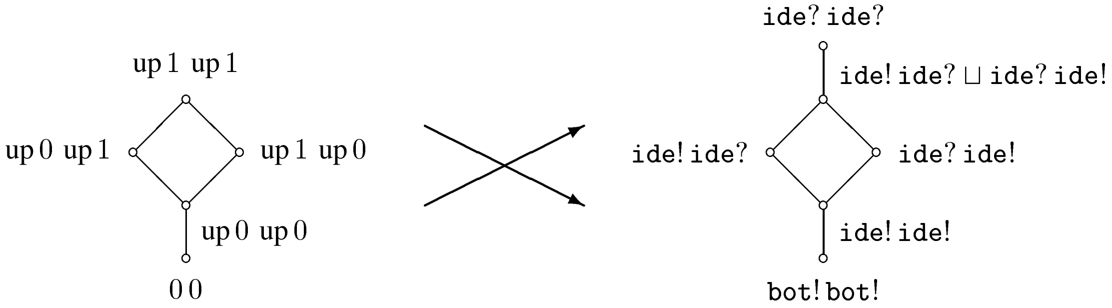
It is surprising that we obtain the same results by asking seemingly unrelated questions. In the forward direction we ask 'is the result undefined if the argument or a part of it is undefined' whereas in the backward direction we ask 'must the argument or a part of it be defined, so that the result is defined'. Logically, the latter question is just the contraposition of the definition of lift strictness.

$$\begin{array}{ll} \text{forward analysis:} & (\forall x)\ x = \bot \Longrightarrow \varphi(x) = \bot \\ \text{backward analysis:} & (\forall x)\ \varphi(x) \neq \bot \Longrightarrow x \neq \bot \end{array}$$

As a first step towards relating the two directions of analysis we map the abstract forward domains to the abstract domains of backward analysis. Semantically, the mapping is just the set-theoretic complement mapping Scott-closed to Scott-open sets whence the use of the operator $\overline{\cdot}$.

$$\begin{array}{lll} \overline{0} = \mathtt{ide} & \overline{0} = \mathtt{ide?} & \\ \overline{1} = \mathtt{bot} & \overline{\mathrm{up}\, a} = \overline{a}! & \overline{a_1 \ldots a_n} = \bigsqcup\{\, \overline{b_1} \ldots \overline{b_n} \mid a \sqsubseteq b \,\} \end{array}$$

The diagram below illustrates the translation of products. Note that we simplified the context expressions as much as possible.



The context lattice is generally an order of magnitude larger. The forward lattice for 7-tuples, for example, contains $2^7 + 1 = 129$ elements while the backward lattice contains the impressive number of 2.414.682.040.998 elements, see [45]. Order-theoretically, the backward lattice amounts to the set of all order ideals of the forward lattice. Using the translation we may relate forward and backward analysis by

$$\mathcal{E}^{\mathrm{f}}[\![e]\!]\ a \sqsubseteq b \Longleftrightarrow \mathcal{E}^\flat[\![e]\!]\ \overline{b} \sqsubseteq \overline{a},$$

which shows that the two directions of analysis have exactly the same power. So, instead of asking $\mathcal{E}^{\mathrm{f}}[\![e]\!]\ (\mathrm{up}\, 0)\ (\mathrm{up}\, 1) \sqsubseteq 0$ we may pose the question $\mathcal{E}^\flat[\![e]\!]\ \mathtt{ide} \sqsubseteq \mathtt{ide!}\ \mathtt{ide?}$ which is guaranteed to give the same result and vice versa. Even with respect to the run-time complexity both directions of analysis are equivalent. If we ignore the trivial cases ($\varphi^{\mathsf{f}}\,0\ldots0$ and $\varphi^{\flat}\,\mathtt{bot}$) forward analysis seeks for a fixpoint in the set of monotone functions from A to $\mathbf{2}$ while backward analysis seeks for a fixpoint in the isomorphic set of all order ideals over A.

The reader familiar with projection-based analysis has certainly noticed that the definition of $\mathcal{E}^{\flat}$ is biased towards the above equivalence result. We have, for example, made no use of the abstract element $\mathtt{bot}?$ (pronounce 'absent') which can be used to detect the absence of arguments.

$$\begin{array}{lll} \varphi^{\flat}\,\mathtt{ide} = \mathtt{bot}?\,\mathtt{ide}? & \Longrightarrow & \varphi \text{ ignores its 1. argument} \\ \varphi^{\flat}\,\mathtt{ide} = \mathtt{ide}?\,\mathtt{bot}? & \Longrightarrow & \varphi \text{ ignores its 2. argument} \end{array}$$

The context $\mathtt{bot}?$ maps every argument to the recipe which does not terminate if forced. If this is possible without altering the meaning of the function call, φ obviously does not inspect the argument in question. The absence of arguments is an example for a property not expressible in an ideal-based analysis. The proof is simple: Since any ideal-based analysis can only characterize properties that are downward closed, it suffices to exhibit two functions $\varphi_1 \sqsubseteq \varphi_2$ such that φ_2 ignores its argument while φ_1 does not. Clearly, $\varphi_i : \mathbf{2}_{\perp} \multimap \mathbf{2}$ with $\varphi_1(a) = \textbf{down}\,a$ and $\varphi_2(a) = \top$ do the job.

Using the context $\mathtt{bot}?$ we can improve the definition of $\mathcal{E}^{\flat}$ by setting arguments which are not required to $\mathtt{bot}?$ instead of $\mathtt{ide}?$.

$$\begin{array}{rcl} \mathcal{E}^{\flat}[\![x_i]\!]\,\kappa & = & \mathtt{bot}?\ldots\kappa\ldots\mathtt{bot}? \\ \mathcal{E}^{\flat}[\![c]\!]\,\kappa & = & \mathtt{bot}?\ldots\mathtt{bot}? \end{array}$$

Especially the analysis of $\mathtt{freeze}\,e$ profits from the introduction of $\mathtt{bot}?$. First note that the precise meaning of $\kappa?$ is 'the value may or may not be required, but if it is then κ's worth of it will be needed'. Hence, if the demand on $\mathtt{freeze}\,e$ is $\kappa?$ we analyse e with respect to κ and adjust the result by setting the passing modes of the arguments to call-by-need. Formally, the abstract interpretation of $\mathtt{freeze}\,e$ is

$$\begin{array}{rcl} \mathcal{E}^{\mathsf{f}}[\![\mathtt{freeze}\ e]\!]\,(\kappa!) & = & \mathcal{E}^{\flat}[\![e]\!]\,\kappa \\ \mathcal{E}^{\mathsf{f}}[\![\mathtt{freeze}\ e]\!]\,(\kappa?) & = & \mathcal{E}^{\flat}[\![e]\!]\,\kappa \sqcup \mathtt{bot}?\ldots\mathtt{bot}?. \end{array}$$

This amounts to the guard operation mentioned above (cf to the analysis of `reverse`).

To summarize: We have seen that the expressivity of an analysis is less a matter of direction (forward or backward) but more a matter of abstract domains (ideals or projections). An ideal-based analysis can neither detect absence of arguments nor their head strictness[10]. Especially the inability to find head strictness is a major drawback since many list-processing functions are head strict or head strict in a head strict context. Furthermore, we have seen that head strictness is more relevant than tail strictness to the generation of code.

1.3 Historical outline

At the time of writing there are several hundred papers which are more or less relevant to strictness analysis. It is therefore useful to introduce some terminology for classifying strictness analysis methods. The following classification is partly due to Davis and Wadler [51] and to Deutsch [53].

[10] To be precise an ideal-based analysis cannot detect isolated head strictness as opposed to combined head and tail strictness.

First-order vs. higher-order Functions which operate on functions are called higher-order contrasting them to the more usual first-order functions which operate on data. Analysis techniques such as the one presented in this dissertation may be restricted to first-order programs.

Scalar vs. structured values An analysis technique may give poor results for functions working on structured values like lists or trees. The technique on which our development is based was largely motivated by the need to detect, for example, head strictness on lists. Alternative terms for this classification include flat vs. non-flat and discrete vs. non-discrete.

Monomorphic vs. polymorphic Most modern functional languages like MIRANDA are based on the Hindley-Milner type system [115] featuring polymorphic types and functions. While this offers considerable flexibility to the user it adds extra problems to the analysis of strictness. A large amount of work is, in fact, dedicated to the solution of these problems.

Ideal-based vs. projection-based vs. . . . Different techniques use different abstract values to encode strictness information. The 'classical' analysis of Mycroft [117], for example, is based on ideals (or Scott-closed sets) while our technique as the title of the dissertation suggests is based on projections. Other choices of abstract values include paths (sequences of bound variables) and PERs (partial equivalence relations). While this classification refers to the non-standard semantics one may also distinguish between different representations of abstract values. As one of the main contributions of this dissertation we present a compact representation of backward abstractions.

Forward vs. backward A forward analysis tries to infer information about the result of a program given information about its input. Conversely, a backward analysis determines information about the input of a program given information about the output. Traditionally, ideal-based strictness analysis is forward while projection-based analysis is backward. [We keep this tradition.] However, these combinations are by no means compelling. Hughes and Launchbury have shown that it is possible to reverse an arbitrary analysis [86] and there is always a best reversal. This does not imply that the reversal carries the same information as the original abstraction. An earlier paper [85] demonstrates, in fact, that there is "a definite bias towards backwards analysis". Finally, note that an analysis may propagate information in both directions. A typical example is a projection-based higher-order analysis: An abstract value in the analysis consists of a backward component (the usual one) and a forward component for functional values.

High vs. low fidelity An analysis is termed low fidelity if an n-ary function is only analysed separately in each argument. It is called high fidelity if additionally all possible combinations of arguments are considered. A high fidelity analyser can detect joint strictness in two or more parameters (the conditional is, for example, joint strict in the second and third parameter) while a low fidelity procedure cannot. Closely related is the distinction between *relational* and *independent* analysis which is concerned with the treatment of tuples. An independent analysis abstracts a tuple to a tuple of abstract values while a relational analysis employs, roughly speaking, sets of abstract tuples. Semantically, this corresponds to the notion of tensor product [120, 124]. Of course, a strictness analysis may switch between the two concepts depending on the precision wanted. If we identify a n-ary function with an unary function taking a n-tuple, then a high fidelity analysis is *locally relational* with respect to the function arguments. Since the non-standard semantics we present yields the least abstraction of a function it is a relational 'analysis'. However, due to complexity considerations our approximation semantics is independent.

On-line vs. global An analysis is called on-line if a function is analysed only once, that is, independent of its invocation contexts. Conversely, a global algorithm re-analyses each procedure for every context.[11] A nice example of an on-line algorithm is the Milner-style type-inference [115]. A function definition is analysed once producing a so-called principal type. Each application of the function is then typed by instantiating the principal type. It is immediate that on-line algorithms go well together with separate compilation and module systems while global algorithms do not. The generic analysis presented in this dissertation is certainly a first step towards an on-line algorithm for projection-based strictness analysis.

Denotational vs. operational An analysis must be proven correct with respect to some 'standard' semantics. Naturally, the choice of the standard semantics depends on the properties we wish to prove. If the analysis gives information about the order of evaluation, then a (standard) denotational semantics is certainly a bad choice. In this particular case an operational semantics may serve as an adequate reference point. Nonetheless, a majority of the approaches described in the literature (our approach being no exception) refer to the denotational semantics. The main reason is probably that a denotational semantics is considered as normative (see [142]). Furthermore, the proof of correctness is likely to be easier since a denotational semantics is more abstract than an operational one.

A related classification is due to Sestoft [140] who distinguishes between *extensional* and *intensional* analysis algorithms. An analysis of the former type only finds information about the input-output behaviour of a program while an intensional analysis may infer information about the run of a program. Sestoft points out that an intensional analysis must not necessarily refer to a denotational semantics and vice versa (though it is likely to be the case).

Before turning to the historical outline let us point out that the terminology introduced is by no means established. It is quite common, for example, to use the term 'abstract interpretation' synonymously for ideal-based analysis. This is rather unfortunate since projection-based analysis fits equally well into the framework of abstract interpretation. Furthermore, forward is often identified with ideal-based and conversely backward with projection-based analysis.

The historical outline is structured as follows. We start with a rather detailed account of the work on forward analysis. Second, the development of backward analysis is sketched. Finally, we give some pointers to alternative approaches.

Basic and introductory articles The theory of abstract interpretation originates in the work of Patrick and Radhia Cousot [39, 40] on the analysis of flow-chart programs. A general framework in a denotational setting was developed by Nielson in a series of papers, see [121, 122], for example.

An easily accessible account of the "compile-time analysis of functional programs" is given in [82]. For a more technical introduction focusing on theoretical foundations see [11].

[11] The definition given in Deutsch [53] is actually more demanding in that an on-line algorithm must furthermore be able to distinguish between an infinite number of abstract contexts. We decided not to be that restrictive.

There is even a german textbook [144] dealing among other things with the topic of abstract interpretation.

History of forward analysis Mycroft was the first who employed the framework of abstract interpretation to the analysis of functional programming languages [117]. In *loc. cit.* he describes a strictness analyser for a first-order, monomorphic language with flat domains which was sketched in Section 1.1. It is interesting to note that the strictness analysis is complemented by a termination analysis ("[...] we use the two non-standard interpretations to 'sandwich' the standard interpretation [...]"). Burn, Hankin, and Abramsky extended the analysis in a natural way to higher-order programs by abstracting the function space $D_1 \to D_2$ to $A_1 \to A_2$ where A_i is the abstraction of D_i ([30] is a revised paper well worth reading). The analysis is often referred to as BHA-style analysis. Abramsky [9] subsequently showed that strictness is a *polymorphic invariant* which means that every instance of a polymorphic function exhibits the same strictness properties suggesting a way to deal with polymorphic functions. The result was obtained using an operational semantics. Abramsky and Jensen [12] re-established the result by interpreting polymorphic functions as transformations between 'relators'.

The papers cited are mainly concerned with the theoretical foundations of the analysis, that is, their proof of correctness. In [30] a Galois connection (or adjunction) is established between the concrete and the abstract domains which entails the use of powerdomains (in *loc. cit.* the Hoare powerdomain is used). Mycroft and Jones proposed an alternative approach based on logical relations obviating the need for powerdomains [119]. Abramsky refined this idea in [7] and proved the equivalence of the two techniques.

Since non-trivial properties of computable functions are undecidable, any analysis must be incomplete, that is, it must fail to report some strictness properties. It is therefore natural to ask how much information is lost by a particular approach. It was only in 1991 that Sekar *et al.* [138] showed that Mycroft's analysis and a method of their own [139] are complete in the following sense: Every property is found which does not depend on the distinction between constants. Reddy and Kamin [134] generalized the results to a broader class of abstract interpretations by recasting the operational arguments of Sekar *et al.* in a denotational setting. A first step towards a more general framework is made in [118].

On the negative side, Kamin [101] showed that an ideal-based analysis like BHA-style analysis cannot determine head strictness. His result is in fact more general, namely, that any *monotonic* abstract interpretation on domains of finite height is unable to characterize head strictness. It is interesting to note that this immediately implies that a projection-based analysis is not monotonic! We will return to this issue in Section 5.6. Another example for a non-monotonic analysis is Hunt's PER-based approach [89].

The question of complexity was first addressed by Hudak and Young [76] who showed that first-order strictness analysis is DEXPTIME-complete.[12] More precisely, they proved that evaluating a recursively defined monotone Boolean function at any point is complete in deterministic exponential time in the length of the function definition. The situation is, however, not completely lost since user-defined functions are unlikely to cause such a worst-case behaviour. It is therefore assumed that strictness analysis is practicable (at least in the first-order case) and, in fact, a number of algorithms with a good average-case behaviour have been devised (see below).

Among the first who considered questions of efficiency were Clack and Peyton Jones [35]

[12] The result is due to Albert Meyer.

who addressed in *loc. cit.* the problem of finding fixpoints of recursively defined monotone Boolean functions. An essential prerequisite of a good average-case algorithm is a compact representation of monotone functions which allows for a simple test of equality. Clack and Peyton Jones devised for this purpose the frontier representation. The frontier of a monotone function $\varphi : \mathbf{2}^n \to \mathbf{2}$ is given by the set $\mathrm{Max}(\varphi^{-1}(0))$ which represents the demarcation line between $\varphi^{-1}(0)$ and $\varphi^{-1}(1)$.[13] Martin and Hankin [113] generalized the frontier algorithm to higher-order functions over arbitrary finite lattices. Later work [88] improved on the presentation of the algorithm using the correspondence between frontiers and ideals. Based on the work of Hughes [80] and Baraki [16], who showed how to compute any instance of an abstract polymorphic function from its simplest instance[14], Seward extended the frontiers algorithms to deal with polymorphic functions [141].

The case is, however, far from being closed. It was soon realized that it is infeasible to compute the complete fixpoint of a higher-order function due to the rapid growth of the abstract domains (at least exponential in the size of the type). Hunt and Hankin [66] proposed to sacrifice accuracy for efficiency by calculating only safe approximations to the exact fixpoint. Another line of research was initiated by Rosendahl [135] who generalized demand-driven or chaotic fixpoint iteration due to the Cousots [38] to handle higher-order functions. Demand-driven fixpoint iteration allows to calculate the fixpoint only for a small set of arguments. If, for example, we are only interested in simple strictness (joint strictness is of no interest wrt the generation of code) then only calls of the form $\varphi^{\mathrm{f}}\,(\mathrm{up}\,1)\ldots(\mathrm{up}\,0)\ldots(\mathrm{up}\,1)$ are of interest. Based on this approach a prototype analyser has been build for HASKELL [91]. It should be noted, however, that demand-driven fixpoint iteration is a global algorithm which does not interact with separate compilation.

Only recently more syntactic methods have been proposed to cope with the problem of combinatorial explosion. Chuang and Goldberg [34] represent abstract functions by λ-expressions in a certain normalform, Ferguson and Hughes [57] use sequential algorithms and Mauborgne [114] encodes higher-order functions into Typed Decision Graphs, a refinement of Ordered Binary Decision Diagrams.

History of backward analysis The first backward analyser was developed as early as 1981 by Johnson [94] and may be considered as the pendant to Mycroft's work.[15] Four years later Wray [152] implemented an analyser for a second-order language which could detect strictness and absence of arguments. At the same time Hughes [78] published the precursor to our work: a method for analysing functions on structured values like lists, trees etc. In retrospective the paper was deficient in two respects. First, no theoretical foundation was given and second, the analyser relied on rather ad-hoc symbolic manipulations. A successor paper [79] which tried to remedy this situation explained contexts as continuations and proved the analysis correct with respect to a collecting continuation semantics. A more satisfactory explanation was discovered by Wadler and described in a joint paper with Hughes [150]. The key insight was that the operational notion of evaluation could be captured by the domain-theoretic notion of projection allowing for a simpler correctness proof (with respect to the direct semantics) and for a more systematic manipulation of contexts. As the analysis was restricted to a first-order monomorphic language a series of papers followed which extended the analysis in the obvious directions.

[13] More precisely this amounts to the maximum-0-frontier.

[14] The method devised gives only exact results for first-order functions. In the higher-order case minor approximations are introduced.

[15] It is worth noting, however, that Mycroft's analysis is high fidelity while Johnson's is low fidelity.

In [87] Hughes generalizes the ideas of Wray to higher-order functions giving a general framework for backward analysis. Building on the view of polymorphic functions as natural transformations the same author adapted the polymorphic invariance results of Abramsky to projection-based analysis [81]. We will elaborate on this topic in Chapters 6 and 7 (albeit on a more syntactical level). These ideas finally led to a prototype implementation for a first-order subset of HASKELL [105]. It is mainly the last work which inspired us to pursue the topic of strictness analysis.

A different line of research was followed by Davis and Wadler [50] who presented a high fidelity backward analyser in *loc. cit.* However, upon inspection the analyser is really a forward analyser at a higher level of abstraction! The abstract values manipulated are projection transformers and the abstraction of a function is a projection transformer transformer. A successor paper [51] further developed the theme of low vs. high fidelity in various 'dimensions'. As we shall see in Chapter 4 the additional level of abstraction is not necessary to obtain a high fidelity or even a relational analysis.

Davis was the first to address questions of completeness and optimality. He shows in [46] that a stable function possesses a least abstraction and that the function—provided it is strict—is in turn determined by its least abstraction.[16] Unfortunately, this paper and its successor [47] suffer from a number of errors. The most far-reaching one is probably the misconception that there is no least backward abstraction of 'parallel or'. We show the contrary in Chapter 5 namely that every first-order function has a least abstraction. Nonetheless, it is fair to say that Davis asked the right questions thereby giving a great impetus to our work.

There are a number of related approaches which may be semantically characterized as projection-based. Dybjer [54] attempted to explain backward analysis as *inverse image analysis* of Scott-open sets or filters (see [55] for a revised version). Since filters correspond to smash projections, inverse image analysis is really a special case of projection-based analysis. Examples for non-smash projections are '`bot?`' and 'H'. Nonetheless the cited paper is notable for its mathematical elegance being due to the heavy use of algebraic reasoning and for the first step towards relating forward and backward analysis. The latter aspect was further developed by Chuang and Goldberg [33] who showed how to reverse an arbitrary ideal-based non-standard semantics building on the duality of ideals and filters. They apply the method to BHA-analysis thereby obtaining a relational higher-order backward analysis.

A more practically oriented approach was described by Hall and Wise in [65]. Their strictness patterns denote projections on the infinite domains of LISP-like lists. It is worth noting that Hall and Wise were among the first to study the utilization of strictness information in a code generator (they propose to create different versions of a function for each actual context in which the function is called). Jones and Le Métayer [97] introduce the similar notion of necessity patterns which are in fact a restricted form of strictness patterns.

Last but not least Burn developed in a series of papers [28, 29, 31, 32] the 'evaluation transformer model of reduction'. Burn was inspired by the work of Hughes [78] and Wadler [148]. Combining some of their ideas he shows in [28] how to derive an evaluation transformer from Wadler's forward analysis of lists. Hence Burn was probably the first to invert (the results of) an analysis. The relation between projection-based analysis and Burn's evaluation transformers was unclear for some time. The author attributes this to the somehow informal definition of evaluators. As matters stand evaluators may safely be identified with projections and evaluation transformers with projection transformers the former notions re-

[16] Since Davis considers functions of type $D_\perp \multimap E_\perp$ instead of $D_\perp \multimap E$ as we do, he additionally requires the functions to be bottom-reflecting. In fact the set of strict, bottom-reflecting functions from $D_\perp$ to $E_\perp$ is isomorphic to the function space $D_\perp \multimap E$. For obvious reasons we prefer to work with the latter space.

ferring to the operational semantics and the latter notions to the denotational one. More precisely, as Burn's analysis was initially based on the ideal-based approach his evaluators are identified as smash projections. It is worth noting that smash projections are 'easy' to implement since the complete evaluation is finished upon return (contrary, for example, to the projection 'H').

Strictness analysis by type inference Both in forward and in backward analysis the behaviour of a function is coded into an abstract function. If a binary function, say, φ is strict in both arguments we have $\varphi^{\mathrm{f}}\,(\mathrm{up}\,0)\,(\mathrm{up}\,1) = 0$, $\varphi^{\mathrm{f}}\,(\mathrm{up}\,1)\,(\mathrm{up}\,0) = 0$ and $\varphi^{\flat}\,\mathrm{ide} = \mathrm{ide}!\,\mathrm{ide}!$. With a little imagination we may re-interpret the abstract functions as specifications using non-standard types: The forward abstraction may be rewritten as $\varphi :: ((\mathrm{up}\,0)\,(\mathrm{up}\,1) \to 0) \wedge ((\mathrm{up}\,1)\,(\mathrm{up}\,0) \to 0)$. Accordingly, the backward abstraction reads as $\varphi :: \mathrm{ide}!\,\mathrm{ide}! \to \mathrm{ide}$. This interpretation is not as far-fetched as one might think since both forward values, ideals, and backward values, projections, have been used to model types, see [112] and [111]. Building on the above interpretation strictness analysis is then viewed as non-standard type inference. Note that the underlying type system naturally involves subtypes as $0 \sqsubseteq 1$ and $\sigma_1! \sqsubseteq \sigma_2?$ if $\sigma_1 \sqsubseteq \sigma_2$.

This line of research was initiated by Kuo and Mishra [106, 107] drawing on earlier work [59] on polymorphic type-inference in the presence of subtypes. The authors claim in [107] that their technique "improves on previous work as it does not require fixpoint iteration". However, it was soon realized that their type system gives strictly weaker information than BHA-style abstract interpretation. Jensen [92] showed that an equally powerful system is obtained by adding conjunctive types shedding some doubt on the thesis of Kuo and Mishra.[17] An equivalent system was developed independently by Benton [18]. In *loc. cit.* he proves that the 'strictness logic' is correct with respect to the standard semantics and shows that it is polymorphically invariant. BHA-style analysis and consequently the systems of Jensen and Benton are independent analyses since products are abstracted to products of abstract values. Strictness logic can be made relational by adding disjunctive types. Corresponding extensions are studied in [17] and [93]. Using disjunctive types bistrictness (strictness in both arguments) may be specified by $(\mathrm{up}\,0)\,(\mathrm{up}\,1) \vee (\mathrm{up}\,1)\,(\mathrm{up}\,0) \to 0$. Benton [19] further extended the logic to languages including monomorphic lists.

The work of Jensen and Benton concentrates mainly on the theoretical foundations of strictness logic. Practical questions are addressed by Hankin and Le Métayer who showed in [67, 68] how to derive an algorithm (specified by an abstract machine) from a type system. Lackner Solberg [108] applied their technique to an inference system which performs both strictness and totality (termination) analysis ([109] is a precursor to that work).

An alternative system which bears close resemblance to the 'type interpretation' of backward abstractions (see above) was proposed by Wright [153]. Instead of forming types from 0, 1, up and $\to$ Wright uses two versions of the type constructor $\to$ corresponding to strict and non-strict functions.[18] This idea was further developed by Amtoft [13, 14] and Henglein [72]. While Wright uses unification over the theory of Boolean rings to solve type constraints arising in his inference algorithm, Amtoft and Henglein resort to standard fixpoint techniques.

[17] The example above shows that conjunctive types are necessary to express strictness in two arguments.

[18] Actually, Wright analyses the intensional notion of neededness of which strictness is an approximation.

Path analysis Strictness is an extensional property referring to the input-output behaviour of a function. If a function, say, $\varphi\, x_1 \ldots x_n$ is identified as strict in the i-th argument, then we know that the variable x_i is accessed during the evaluation of the function body. However, with respect to the generation of code it is also interesting to know whether a variable was or wasn't accessed before a given point in the execution and whether a variable will or won't be accessed after a given point. To see why this information is useful a brief explanation of the internal workings of `freeze` and `unfreeze` may prove helpful. The operation `freeze` e builds a recipe which contains necessary information to resume the computation of e at a later point in the execution. When the recipe is forced the first time the value of e is computed and the recipe is subsequently updated with the computed value. Thus when a recipe is forced a second time the value is already at hand. To distinguish between the different modes some form of bookkeeping is required. Typically, the status of a recipe is recorded in a tag bit. Now, if we know about the past of a recipe the status check at run-time may be saved. Accordingly, if we know that a recipe is not forced a second time the update operation may be saved. Optimizations of this kind require an intensional analysis giving information about the run of a program.

Bloss and Hudak [24] were the first to address questions about the order of evaluation. In *loc. cit.* four different analyses were specified giving information about the definite (non-) evaluation of expressions in the past (future). A later paper [23] generalized these analyses to a so-called path semantics. A path through a function is an ordering on the evaluation of the function's argument. The path semantics maps an expression to a set of possible paths through the expression (actually powerdomains are used). The analysis given in *loc. cit.* was restricted to a monomorphic, first-order language with scalar types. An extension to higher-order languages is described in [22]. Gomard and Sestoft [62, 63] presented a path analysis that is able to deal with recursive data structures. Interestingly, their analysis works backward yielding results similar to ours.

1.4 Plan of the dissertation

Projection-based strictness analysis is often put in opposition to abstract interpretation. We feel that this view is by no means justified. On the contrary, backward analysis fits very well into the scheme of abstract interpretation depicted in Figure 1.1. In fact, our dissertation is essentially structured according to this scheme, see Figure 1.6.

Chapter 2 introduces the syntax of a first-order, polymorphic language with algebraic data types. This includes various transformations such as the call-by-value transformation which aim at simplifying the presentation of the various semantics.

A denotational semantics of the language respectively its strict, lifted variant is given in Chapter 3. Note that the presentation assumes elementary knowledge of domain theory. In order to keep the dissertation self-contained we have listed the necessary prerequisites including notation and nomenclature in Appendix A.

Chapter 4 and Chapter 5 jointly introduce the non-standard semantics. We have seen that contexts may be represented by projections, a concept borrowed from domain theory. Chapter 4 develops a theory of projections ranging from elementary characteristics over questions of representability to algebraic properties. Building on this theory we show in Chapter 5 that every continuous, first-order function possesses a least abstraction. For the first-order language a compositional non-standard semantics is given which maps a textual function to its least abstraction.

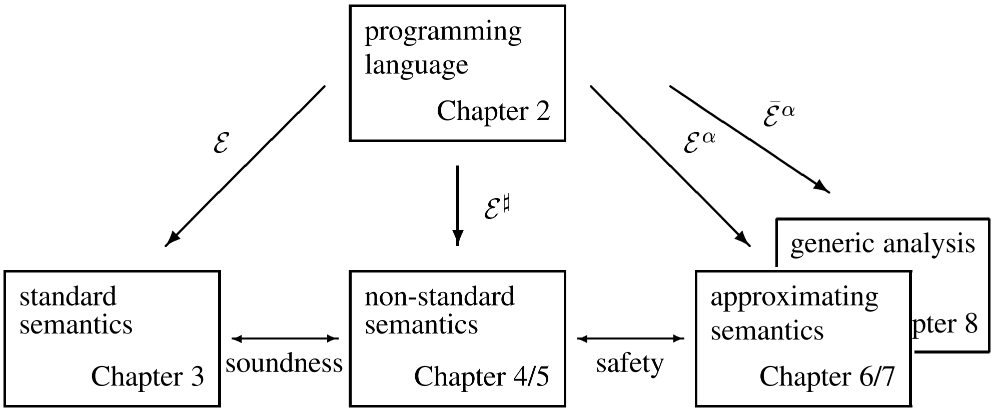


Figure 1.6: Structure of the dissertation

Chapter 6 and Chapter 7 mirror the two preceding ones as they define the abstract variants of projections, called contexts for lack of names, and of least abstractions. Chapter 6 discusses in particular the representation of projections on polymorphic and on recursive types which are not treated explicitly in the non-standard semantics. The formal definition of the approximation semantics in Chapter 7 is complemented by numerous examples demonstrating the analytical power of backward analysis.

Chapter 8 develops the theme of generic backward analysis which may be considered as a smart realization of the approximation semantics.

As already noted, Appendix A comprises some basic material of order and domain theory. Finally, Appendix B provides the source code of all types and functions used in this work.

One of the symptoms of approaching nervous breakdown is the belief that one's work is terribly important, and that to take a holiday would bring all kinds of disaster.

— Bertrand Russell, *The Conquest of Happiness (1930) ch. 5*

Chapter 2

Syntax

If we spoke a different language,
we would perceive a somewhat different world.
— Ludwig Wittgenstein

Subject of the analysis is a first-order, polymorphic, non-strict functional language incorporating user-defined data types. This chapter introduces the syntax of the language; many examples of its use are to be found in Appendix B. In order to simplify the presentation of the various semantics the source language first undergoes a series of transformation steps. The resulting language, a *strict* language with *de Bruijn indices*, is then used as the basis for the chapters to follow. Since the source language as well as the transformation steps are fairly standard—though probably not in the context of program analysis—the cognoscenti map safely skip this chapter.

Plan of the chapter Section 2.1 presents the concrete syntax of the functional programming language. The core of the language, that is, its abstract syntax is identified in Section 2.2. Section 2.3 introduces a source-to-source transformation which makes explicit where expressions are delayed and where delayed expressions are to be evaluated. Another transformation which replaces variables by de Bruijn indices is described in Section 2.4.

2.1 Concrete syntax

Table 2.1 gives the concrete syntax of the source language in EBNF. Many examples are contained in Appendix B which provides and explains the source code of all types and functions used in this work. Therefore we confine ourselves to a few remarks.

Programs consist of type definitions followed by a sequence of function specifications and definitions. Type definitions specifying so-called algebraic data types are a common feature of most modern functional programming languages. Here are a few examples including types which are usually thought of as 'built in'.[1]

[1] Note that we make liberal use of infix and mixfix notation whenever this conforms to the common standard of functional programming languages. Note, too, that similar syntax is used both for types and values: `(num,num)` denotes the type of all pairs with numerical components while `(1,2)` represents an element of that type.

production			comment
⟨prg⟩	$\longrightarrow$	{ ⟨tdef⟩ } { ⟨spec⟩ ⟨fdef⟩ }	program
⟨tdef⟩	$\longrightarrow$	⟨tcon⟩ { ⟨tvar⟩ } ::= ⟨constructs⟩	data type definition
⟨constructs⟩	$\longrightarrow$	⟨con⟩ { ⟨texp⟩ } { $\mid$ ⟨con⟩ { ⟨texp⟩ } }	constructs
⟨texp⟩	$\longrightarrow$	⟨tvar⟩	generic type variable
	$\mid$	⟨tsys⟩	system-defined type
	$\mid$	⟨tcon⟩ { ⟨texp⟩ }	application of a data type
	$\mid$	(⟨texp⟩)	parenthesised type expression
⟨spec⟩	$\longrightarrow$	⟨fun⟩ :: { ⟨texp⟩ } $\to$ ⟨texp⟩	function specification
⟨fdef⟩	$\longrightarrow$	⟨fun⟩ { ⟨var⟩ } = ⟨exp⟩	function definition
⟨exp⟩	$\longrightarrow$	⟨var⟩	variable
	$\mid$	⟨sys⟩ { ⟨exp⟩ }	system function call
	$\mid$	⟨con⟩ { ⟨exp⟩ }	constructed value
	$\mid$	⟨fun⟩ { ⟨exp⟩ }	user function call
	$\mid$	<u>if</u> ⟨exp⟩ <u>then</u> ⟨exp⟩ <u>else</u> ⟨exp⟩	conditional
	$\mid$	<u>case</u> ⟨exp⟩ <u>of</u> ⟨alt⟩ { $\mid$ ⟨alt⟩ }	case analysis
	$\mid$	<u>let</u> ⟨def⟩ { ; ⟨def⟩ } <u>in</u> ⟨exp⟩	local value definition
	$\mid$	<u>letrec</u> ⟨def⟩ { ; ⟨def⟩ } <u>in</u> ⟨exp⟩	recursive value definition
	$\mid$	(⟨exp⟩)	parenthesised expression
⟨alt⟩	$\longrightarrow$	⟨con⟩ { ⟨var⟩ } $\to$ ⟨exp⟩	alternative
⟨def⟩	$\longrightarrow$	⟨var⟩ = ⟨exp⟩	binding

Table 2.1: Concrete syntax of the language

```
bool      ::= True | False
(α1,α2)   ::= (α1,α2)
[α]       ::= [] | α:[α]
tree α    ::= Empty | Node (tree α) α (tree α)
```

A type definition introduces one or more constructors each having zero or more arguments. A constructor must not occur in more than one definition. Accordingly, the type parameters of an algebraic data type must be pairwise different. Some terminology is helpful when discussing structured types. If a type definition contains only nullary constructors the type is called *enumeration type*. A data type is termed *tuple type* if it consists only of a single constructor. This term is semantically justified since a one element sum is isomorphic to its single component being a product in the case of data types. If a type is parameterized with respect to one or more type variables we qualify the type as *polymorphic*. Hence, the last three definitions introduce polymorphic pairs, lists, and trees, respectively.

Elements of a data type are built with the constructors listed in the type definition. The function `fulltree` a n, for example, constructs a full binary tree of depth n, labelling all nodes with a.

```
fulltree :: α num → tree α
fulltree a n = if n=0 then Empty
               else let t = fulltree a (n-1) in Node t a t
```

The concrete syntax distinguishes between three sorts of functions: predefined functions, constructors, and user-defined functions. To tell the three cases apart we will obey the following convention: predefined functions are given symbolic names like '=' and '-' (furthermore they are written infix), constructors like `Empty` and `Node` start with a capital letter while user-defined functions like `fulltree` are in lower case.

Local value definition are introduced by let expressions. For recursive value definitions letrec $x_1 = e_1; \ldots ; x_n = e_n$ in e is used which behaves exactly like let except that the scope of $x_1, \ldots , x_n$ extends over e as well as over the definientia $e_1, \ldots , e_n$. Note that we deliberately dispense with local function definitions. This is not a severe restriction, however, since local function definitions may be transformed into global ones using a transformation called λ-lifting [95]. In Appendix B we illustrate the technique by means of a simple example. [Appendix B also indicates how list comprehensions, irrefutable patterns and simple uses of higher-order functions may be dealt with.]

Elements of a data type are taken apart by means of a case-expression. The function `depth` calculates the depth of a tree.

```
depth :: tree α → num
depth t = case t of Empty → 0 |
                    Node l a r → 1 + max2 (depth l) (depth r)

max2 :: num num → num
max2 a b = if a>b then a else b
```

A case-expression serves a double purpose: It discriminates between different cases and it introduces names for the components of structured values. The constructors in a case-expression must be pairwise different. Accordingly, the patterns may not contain multiple occurrences of the same variable.[2] This restriction applies to the remaining variable binding constructs, function definitions and local value definitions, as well.

The structure of a case-expression usually adheres very close to the definition of the corresponding type. However, we do not require case analysis to be exhaustive. Thus

```
head x = case x of a:w → a
```

constitutes a legal definition. Many modern functional languages which support an equational style of definition offer pattern matching facilities such as multiple nested patterns, overlapping patterns, irrefutable patterns, guards etc. However, since complex patterns may be translated into simple case-expressions (see [149] for a detailed account), we confine ourselves to the latter.

It goes without saying that we consider only well-typed programs. We will not formally define the notion of well-typedness but rather refer to the classical literature on this subject [115, 43].

2.2 Syntactic categories and abstract syntax

The concrete syntax contains many irrelevant details and is redundant is some respects. Furthermore, most of the constructs are not atomic in that they combine two or more primitive concepts. The abstract syntax abstracts from these details and identifies the core concepts of the language.

[2] In MIRANDA patterns may contain repeated variables imposing an equality constraint on the corresponding parts of a value.

The presentation proceeds in two steps. First, the syntactic categories are declared thereby introducing meta variables which are used henceforth to name elements of the respective categories. The categories are then defined by a set of grammar rules. Note that we will largely follow this pattern throughout this work.

The syntactic categories of the language are shown in Table 2.2. The different categories should be self-explanatory. The category of system-defined types **tsys** comprises at least ε (the empty sequence type sometimes termed `unit`), `char`, and `num`.

meta variable		category	comment
α	$\in$	**tvar**	generic type variables
ς	$\in$	**tsys**	system-defined types
β	$\in$	**tcon**	μ-bound type variables
σ	$\in$	**texp**	type expressions
x	$\in$	**var**	variables
s	$\in$	**sys**	system-defined functions
c	$\in$	**con**	constructors
f	$\in$	**fun**	user-defined functions
e	$\in$	**exp**	expressions
p	$\in$	**pat**	patterns
		prg	programs

Table 2.2: Syntactic categories of the language

The abstract syntax of the language is given in Table 2.3. Let us consider the abstract syntax of type expressions first. The definition of polymorphic lists, for example,

$$[\alpha] \;::=\; [\,] \;\mid\; \alpha : [\alpha]$$

may be translated to the type expression $\mu\beta.[\,]\ \varepsilon \mid \alpha:\beta$. The recursion operator allows to convert type definitions into closed type expressions paving the way for the representation of contexts. [A context may then be considered as a type expression incorporating strictness annotations.] Note that we distinguish between μ-bound type variables representing type names and generic type variables indicating polymorphism.

Sequence types are used for the arguments of constructors, empty sequences corresponding to nullary constructors. Note that the empty sequence is classified as system-defined. This separation plays no rôle until Chapter 6, so let us postpone a discussion of this technical detail until then.

Mutually recursive type definitions pose no problems, since type expressions may be arbitrarily nested. The definitions

```
shrub ::= Root node
node  ::= Void | Fork shrub num shrub
```

are transformed by unrolling the indirect recursive calls to

$$\begin{aligned} \text{shrub} &= \mu\beta.\text{Root}\,(\text{Void}\,\varepsilon \mid \text{Fork}\ \beta\ \texttt{num}\ \beta) \\ \text{node} &= \mu\beta.\text{Void}\,\varepsilon \mid \text{Fork}\ (\text{Root}\ \beta)\ \texttt{num}\ (\text{Root}\ \beta). \end{aligned}$$

The sole condition which we have to make concerns the form of the recursive calls, which must be *identical* to the left-hand-side of the respective type definition. Otherwise it is not possible to translate the definition into the form $\mu\beta.\sigma$. Consequently the definitions

production			comment
texp	$::=$	**tvar**	generic type variable
	$\mid$	**tcon**	μ-bound type variable
	$\mid$	**tsys**	system-defined type
	$\mid$	$\mathbf{con}_1\ \mathbf{texp}_1 \mid \cdots \mid \mathbf{con}_n\ \mathbf{texp}_n$	sum type
	$\mid$	$\mathbf{texp}_1 \ldots \mathbf{texp}_n$	sequence type ($n \geqslant 1$)
	$\mid$	$\mu\mathbf{tcon}.\mathbf{texp}$	recursive type
exp	$::=$	**var**	variable
	$\mid$	**sys exp**	system function call
	$\mid$	**con exp**	constructed value
	$\mid$	**fun exp**	user function call
	$\mid$	$\mathbf{exp}_1 \ldots \mathbf{exp}_n$	sequence
	$\mid$	$\underline{\texttt{case}}\ \mathbf{exp}\ \underline{\texttt{of}}\ \mathbf{con}_1\ \mathbf{pat}_1 \rightarrow \mathbf{exp}_1 \mid \cdots \mid \mathbf{con}_n\ \mathbf{pat}_n \rightarrow \mathbf{exp}_n$	case analysis
	$\mid$	$\underline{\texttt{let}}\ \mathbf{pat}_1 = \mathbf{exp}_1\ \underline{\texttt{in}}\ \mathbf{exp}$	local value definition
	$\mid$	$\underline{\texttt{rec}}\ \mathbf{pat}.\mathbf{exp}$	recursion operator
pat	$::=$	$\mathbf{var}_1 \ldots \mathbf{var}_n$	sequence pattern
prg	$::=$	$\mathbf{fun}_1\ \mathbf{pat}_1 = \mathbf{exp}_1; \ldots ; \mathbf{fun}_n\ \mathbf{pat}_n = \mathbf{exp}_n$	program

Table 2.3: Abstract syntax of the language

```
rotate α₁ α₂   ::= Rotate (rotate α₂ α₁)
instantiate α ::= Instantiate (instantiate (list α))
```

are ruled out. Fortunately, this requirement is usually met in practice. Data type definitions satisfying this property are termed *uniform definitions*, see [105].

In order to formalize the expansion of type expressions we need the concept of an environment. An environment is a finite function mapping identifiers to associated values. We use $\varnothing$ to denote the empty environment. To overlay an environment with a new binding the notation $\cdot[\cdot \mapsto \cdot]$ is employed.

$$\begin{array}{lcll} \varrho[x \mapsto v](y) & = & v & \textbf{if } x = y \\ & = & \varrho(y) & \textbf{otherwise} \end{array}$$

The expansion of a type expression containing occurrences of data types is defined in Table 2.4. Life is simple if the type definitions are non-recursive. In this case we may view them as mere abbreviations. Let $t\ \alpha_1 \ldots \alpha_n ::= \sigma$ be a data type definition. The type expression $t\ \sigma_1 \ldots \sigma_n$ is then replaced by σ thereby substituting $\sigma_1, \ldots ,\sigma_n$ for $\alpha_1, \ldots ,\alpha_n$. In order to avoid the infinite unrolling of recursive types the names of expanded types are recorded. If a 'recursive call' is encountered which is identical to the left-hand-side of the respective type definition it is simply replaced by the name of the data type which now acts as a μ-bound type variable. Otherwise the type definition is non-uniform.

The technique of transforming uniform definitions into μ-expressions is by no means special to types. The same technique is often applied to function definitions for reasons of efficiency. The definition of `repeat`, for example,

```
repeat a = a:repeat a
```

expand$[\![\sigma]\!]\ \mu\ \varrho$ converts σ into a closed type expression containing no references to data types. The set μ contains the names of the data types which already have been expanded. The bindings of type parameters are recorded in ϱ.

$$
\begin{array}{rcll}
\text{expand}[\![\mathbf{texp}]\!] & : & \wp(\mathbf{tvar}) \to (\mathbf{tvar} \to \mathbf{texp}) \to \mathbf{texp} & \\
\text{expand}[\![\alpha]\!]\ \mu\ \varrho & = & \varrho(\alpha) \quad \textbf{if } \alpha \in \textbf{dom}(\varrho) & \\
 & = & \alpha \quad \textbf{otherwise} & \\
\text{expand}[\![\varsigma]\!]\ \mu\ \varrho & = & \varsigma & \\
\multicolumn{4}{l}{\text{expand}[\![c_1\sigma_1 \mid \cdots \mid c_n\sigma_n]\!]\ \mu\ \varrho} \\
 & = & c_1\text{expand}[\![\sigma_1]\!]\ \mu\ \varrho \mid \cdots \mid c_n\text{expand}[\![\sigma_n]\!]\ \mu\ \varrho & \\
\text{expand}[\![\sigma_1 \ldots \sigma_n]\!]\ \mu\ \varrho & = & \text{expand}[\![\sigma_1]\!]\ \mu\ \varrho \ldots \text{expand}[\![\sigma_n]\!]\ \mu\ \varrho & \\
\text{expand}[\![t\ \sigma_1 \ldots \sigma_n]\!]\ \mu\ \varrho & = & \mu t.\text{expand}[\![\sigma]\!]\ (\mu \cup \{t\})\ \varrho' & \textbf{if } t \notin \mu \\
 & = & t & \textbf{if } t \in \mu \wedge (\forall i)\ \sigma_i = \alpha_i \\
 & = & \mathit{error}\ \text{“non-uniform type”} & \textbf{otherwise} \\
 & & \textbf{where} & \\
 & & t\ \alpha_1 \ldots \alpha_n ::= \sigma \text{ is a data type definition} & \\
 & & \varrho' = \varnothing[\alpha_1 \mapsto \text{expand}[\![\sigma_1]\!]\ \mu\ \varrho] \cdots [\alpha_n \mapsto \text{expand}[\![\sigma_n]\!]\ \mu\ \varrho] &
\end{array}
$$

Table 2.4: Expansion of type expressions

may be changed to

```
repeat a = letrec x = a:x in x .
```

The latter definition is more efficient than the first one since it produces a cyclic data structure which consumes only a finite amount of space (see [21, page 186]).

It is perhaps surprising that we may represent mutually recursive definitions using a simple μ-operator. A similar technique is sometimes used to solve simultaneous fixpoint equations (see [133, page 22]).

The transformation of expressions is less involving. Due to the introduction of sequences the syntax of function application is a bit simpler. We will, however, make use of the fact that the arguments of constructors, system- and user-defined functions are always sequences. [The abstract syntax does not allow to tell sequences of length one and single expressions apart. So this is a useful fact to remember.] The conditional `if`-`then`-`else` is no longer primitive as it may be defined in terms of `case`:

$$\underline{\texttt{if}}\ e_1\ \underline{\texttt{then}}\ e_2\ \underline{\texttt{else}}\ e_3 \quad \rightsquigarrow \quad \underline{\texttt{case}}\ e_1\ \underline{\texttt{of}}\ \texttt{True} \to e_2 \mid \texttt{False} \to e_3.$$

Local value definitions and recursive value definitions are transformed according to

$$
\begin{array}{l}
\underline{\texttt{let}}\ x_1 = e_1; \ldots ; x_n = e_n\ \underline{\texttt{in}}\ e \\
\rightsquigarrow\ \ \underline{\texttt{let}}\ x_1 \ldots x_n = e_1 \ldots e_n\ \underline{\texttt{in}}\ e, \\
\underline{\texttt{letrec}}\ x_1 = e_1; \ldots ; x_n = e_n\ \underline{\texttt{in}}\ e \\
\rightsquigarrow\ \ \underline{\texttt{let}}\ x_1 \ldots x_n = \underline{\texttt{rec}}\ x_1 \ldots x_n.e_1 \ldots e_n\ \underline{\texttt{in}}\ e.
\end{array}
$$

Contrary to the μ-operator the recursion operator `rec` allows for multiple bindings since we do not want to duplicate parts of expressions.

2.3 Call-by-value transformation

Call-by-value and call-by-need differ mainly in the kind of objects which are passed to the callee. The former passes values such as numbers, lists etc while the latter passes prescriptions, which tell the called function how to compute the corresponding values. By introducing unevaluated expressions as first-class objects into the language we may confine ourselves to call-by-value as the sole parameter passing mechanism. This in fact very old technique was described by Henderson [71] as a means to build 'lazy evaluation' on top of call-by-value. Table 2.5 introduces the necessary extensions to the abstract syntax. The transformation rules

production			comment
texp	::=	...	
	|	**texp**?	lift type
exp	::=	...	
	|	`freeze` **exp**	delay evaluation
	|	`unfreeze` **exp**	force evaluation

Table 2.5: Extensions to the abstract syntax of the language

are summarized in Table 2.6. Note that not only expressions are subject to transformation but types as well.

The operator `freeze` prevents its argument from being evaluated. The value of `freeze` e is an unevaluated expression, called *recipe* (alternative terms include closure, laze, thunk, or suspension), which may be passed freely between functions, stored in data structures etc. The counterpart of `freeze` is, of course, `unfreeze`. The argument of `unfreeze` must evaluate to a recipe encapsulating an expression, which in turn is evaluated. Thus `unfreeze (freeze` e`)` always denotes the same value as e. With respect to the concrete syntax there are exactly three different places where evaluation of expressions is delayed.

1. The parameters of functions (system-[3] as well as user-defined),
2. the arguments of constructors, and
3. the definientia of local definitions.

Since each item involves sequences, it suffices in fact to delay the elements of a sequence. Conversely, evaluation of a recipe is forced, if a parameter of a function, an argument of a constructor or a locally defined value is accessed, in other words, if a variable is dereferenced.

The distinction between values and unevaluated expressions is also reflected in the type system. The type $\sigma?$, also called lift type for reasons which will become apparent in Section 3.1, comprises unevaluated expressions of type σ. Consequently `freeze` and `unfreeze` possess the following types.

```
freeze   :: α → α?
unfreeze :: α? → α
```

[3]In Chapter 1 we assumed for the sake of simplicity that primitive functions are strict. Here, we decided not be that restrictive.

lazify$\llbracket\sigma\rrbracket$ converts a non-strict type into a strict one.

$$
\begin{array}{rcl}
\text{lazify}\llbracket\textbf{texp}\rrbracket & : & \textbf{texp} \\
\text{lazify}\llbracket\alpha\rrbracket & = & \alpha \\
\text{lazify}\llbracket\beta\rrbracket & = & \beta \\
\text{lazify}\llbracket\varsigma\rrbracket & = & \varsigma \\
\text{lazify}\llbracket c_1\sigma_1 \mid \cdots \mid c_n\sigma_n\rrbracket & = & c_1(\text{lazify}\llbracket\sigma_1\rrbracket) \mid \cdots \mid c_n(\text{lazify}\llbracket\sigma_n\rrbracket) \\
\text{lazify}\llbracket\sigma_1 \ldots \sigma_n\rrbracket & = & (\text{lazify}\llbracket\sigma_1\rrbracket)? \ldots (\text{lazify}\llbracket\sigma_n\rrbracket)? \\
\text{lazify}\llbracket\mu\beta.\sigma\rrbracket & = & \mu\beta.\text{lazify}\llbracket\sigma\rrbracket
\end{array}
$$

lazify$\llbracket e\rrbracket$ makes explicit where the evaluation of expressions in e is delayed and where delayed expressions are forced.

$$
\begin{array}{rcl}
\text{lazify}\llbracket\textbf{exp}\rrbracket & : & \textbf{exp} \\
\text{lazify}\llbracket x\rrbracket & = & \texttt{unfreeze}\ x \\
\text{lazify}\llbracket s\ e\rrbracket & = & s\ (\text{lazify}\llbracket e\rrbracket) \\
\text{lazify}\llbracket c\ e\rrbracket & = & c\ (\text{lazify}\llbracket e\rrbracket) \\
\text{lazify}\llbracket f\ e\rrbracket & = & f\ (\text{lazify}\llbracket e\rrbracket) \\
\text{lazify}\llbracket e_1 \ldots e_n\rrbracket & = & (\texttt{freeze}\ \text{lazify}\llbracket e_1\rrbracket) \ldots (\texttt{freeze}\ \text{lazify}\llbracket e_n\rrbracket) \\
\multicolumn{3}{l}{\text{lazify}\llbracket \underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1\ p_1 \to e_1 \mid \cdots \mid c_n\ p_n \to e_n\rrbracket} \\
 & = & \underline{\texttt{case}}\ \text{lazify}\llbracket e\rrbracket\ \underline{\texttt{of}} \\
 & & c_1\ p_1 \to \text{lazify}\llbracket e_1\rrbracket \mid \cdots \mid c_n\ p_n \to \text{lazify}\llbracket e_n\rrbracket \\
\text{lazify}\llbracket \underline{\texttt{let}}\ p_1 = e_1\ \underline{\texttt{in}}\ e\rrbracket & = & \underline{\texttt{let}}\ p_1 = \text{lazify}\llbracket e_1\rrbracket\ \underline{\texttt{in}}\ \text{lazify}\llbracket e\rrbracket \\
\text{lazify}\llbracket \underline{\texttt{rec}}\ p.e\rrbracket & = & \underline{\texttt{rec}}\ p.\text{lazify}\llbracket e\rrbracket
\end{array}
$$

lazify$\llbracket f_1\ p_1 = e_1; \ldots ; f_n\ p_n = e_n\rrbracket$ converts a given non-strict program into a strict one.

$$
\begin{array}{rcl}
\text{lazify}\llbracket\textbf{prg}\rrbracket & : & \textbf{prg} \\
\multicolumn{3}{l}{\text{lazify}\llbracket f_1\ p_1 = e_1; \ldots ; f_n\ p_n = e_n\rrbracket} \\
 & = & f_1\ p_1 = \text{lazify}\llbracket e_1\rrbracket; \ldots ; f_n\ p_n = \text{lazify}\llbracket e_n\rrbracket
\end{array}
$$

Table 2.6: Transformation of non-strict programs to strict programs

We conclude the section giving an example for the transformation (cf to Chapter 1 for more examples).

```
append x y = case unfreeze x of
                 [] → unfreeze y |
                 a:w → a:freeze (append w y)
```

Note that the overhead of call-by-need compared to call-by-value is directly reflected by the appearance of `freeze` and `unfreeze` in the function body. The example already involves a simple improvement over the basic scheme. We may simplify `freeze (unfreeze` e`)` to e iff e terminates, which in turn is granted if e is a variable. A semantical justification of this transformation is given in Section A.4.1. [The observation that termination properties are useful for the optimization of non-strict languages is again due to Mycroft [117]. It probably comes as no surprise that the analysis of termination properties may be based on projections as well, see [48].]

2.4 De Bruijn transformation

The semantics of an expression, say, e is defined relative to an environment containing bindings for the variables which occur free in e. The expression $x + z$, for example, may be interpreted with respect to the environment $\langle\langle\langle\rangle, x \mapsto 2\rangle, y\ z \mapsto 7\ 65\rangle$ yielding the value 67. Environments are usually modeled by (finite) functions mapping variables to values. Consequently the denotation of an expression possesses the type $(\mathbf{var} \to \mathbf{Val}) \to \mathbf{Val}$. However, for our purposes this is unnecessarily abstract. As in the preceding section we move one step further towards an actual implementation. To this end we factor an environment into a compile-time environment containing only patterns (eg, $\langle\langle\langle\rangle, x\rangle, y\ z\rangle$) and a run-time environment containing the corresponding values (eg, $\langle\langle\langle\rangle, 2\rangle, 7\ 65\rangle$). The compile-time environment is used in the source-to-source translation described below while the run-time environment is employed in the definition of the semantics. The example above shows that a run-time environment boils down to a simple data structure, namely a list of tuples.

The transformation basically replaces variables by access paths into the run-time environment thereby getting rid of all patterns. The expression x+z, for example, translates to 1.1+0.2. An access path is sometimes called de Bruijn index in honour of de Bruijn who invented this technique [52].

The syntactic categories of the de Bruijn variant of our language are shown in Table 2.7. Its abstract syntax is displayed in Table 2.8.

meta variable		category	comment
e	$\in$	**dbe**	de Bruijn expressions
		dbp	de Bruijn programs

Table 2.7: Syntactic categories of de Bruijn's language

production			comment
dbe	::=	$\mathbb{N}.\mathbb{N}$	access path
	|	`freeze` **dbe**	delay evaluation
	|	`unfreeze` **dbe**	force evaluation
	|	**sys dbe**	system function call
	|	**con dbe**	constructed value
	|	**fun dbe**	user function call
	|	$\mathbf{dbe}_1 \ldots \mathbf{dbe}_n$	sequence
	|	`case` **dbe** `of` $\mathbf{con}_1 \to \mathbf{dbe}_1 \mid \cdots \mid \mathbf{con}_n \to \mathbf{dbe}_n$	case analysis
	|	`let` $\mathbf{dbe}_1$ `in` **dbe**	local value definition
	|	`rec` **dbe**	recursion operator
dbp	::=	$\mathbf{fun}_1 = \lambda\mathbf{dbe}_1; \ldots; \mathbf{fun}_n = \lambda\mathbf{dbe}_n$	program

Table 2.8: Abstract syntax of de Bruijn's language

The translation is rather straightforward. An environment containing patterns is maintained on the way down the expression tree. Variable binding constructs like `case` or `let` extend the environment. Variables are searched in the environment and replaced by access paths of the form $k.i$ with $k, i \in \mathbb{N}$. The first number specifies the nesting level, the second number the position of the variable within the pattern at depth k. The variable x, for example, appears at nesting level 1 and position 1 in $\langle\langle\langle\rangle, x\rangle, y\,z\rangle$. The formal syntax of compile-time environments is given in Table 2.9. The transformation rules are summarized in Table 2.10.

production			comment
valenv	::=	$\langle\rangle$	empty environment
	|	$\langle$**valenv**, **pat**$\rangle$	environment layer

Table 2.9: Compile-time value environments

Applying the de Bruijn transformation to our running example yields

```
append = λcase unfreeze 0.1 of
               [] → unfreeze 1.2 |
               :  → 0.1:freeze (append 0.2 1.2).
```

Note that different variables may be mapped to the same de Bruijn index. Since de Bruijn programs are hard to read we will continue to use variables in the examples.

2.5 Bibliographic notes

The concept of a recipe or thunk is nearly as old as the parameter passing mechanism call-by-name. It was invented by Ingerman [90] as a means to implement call-by-name on call-by-value. Henderson [71] introduced it to the world of functional programming. Naturally, thunks also found their way into implementations of lazy functional languages, including among others the lazy SECD machine [71], the Categorical Abstract Machine [37], and the Three-Instruction-Machine [56]. Bloss *et. al.* [25] discuss the representation of thunks and various optimizations based on information about the order of evaluation.

The transformation described in Section 2.4 is due to de Bruijn [52] and was intended for the mechanical evaluation of λ-terms. De Bruijn indices lie also at the heart of the Categorical Abstract Machine. Some authors even base their analysis on a categorical or FP-like notation, see [55] or [85, 86], for example.

access$[\![x]\!]\ \rho\ k$ maps the variable x to an access path in environment ρ at nesting level k.

$$
\begin{array}{rcll}
\text{access}[\![\textbf{var}]\!] & : & \textbf{valenv} \to \mathbb{N} \to \textbf{dbe} & \\
\multicolumn{4}{l}{\text{access}[\![x]\!]\ \langle \rho, x_1 \ldots x_n \rangle\ k} \\
 & = & k.i & \textsf{\textbf{if}}\ x = x_i \\
 & = & \text{access}[\![x]\!]\ \rho\ (k+1) & \textsf{\textbf{otherwise}}
\end{array}
$$

bruijn$[\![e]\!]\ \rho$ is the de Bruijn expression of e with respect to environment ρ.

$$
\begin{array}{rcl}
\text{bruijn}[\![\textbf{exp}]\!] & : & \textbf{valenv} \to \textbf{dbe} \\
\text{bruijn}[\![x]\!]\ \rho & = & \text{access}[\![x]\!]\ \rho\ 0 \\
\text{bruijn}[\![\texttt{freeze}\ e]\!]\ \rho & = & \texttt{freeze}\ (\text{bruijn}[\![e]\!]\ \rho) \\
\text{bruijn}[\![\texttt{unfreeze}\ e]\!]\ \rho & = & \texttt{unfreeze}\ (\text{bruijn}[\![e]\!]\ \rho) \\
\text{bruijn}[\![s\ e]\!]\ \rho & = & s\ (\text{bruijn}[\![e]\!]\ \rho) \\
\text{bruijn}[\![c\ e]\!]\ \rho & = & c\ (\text{bruijn}[\![e]\!]\ \rho) \\
\text{bruijn}[\![f\ e]\!]\ \rho & = & f\ (\text{bruijn}[\![e]\!]\ \rho) \\
\text{bruijn}[\![e_1 \ldots e_n]\!]\ \rho & = & \text{bruijn}[\![e_1]\!]\ \rho \ldots \text{bruijn}[\![e_n]\!]\ \rho \\
\multicolumn{3}{l}{\text{bruijn}[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1\ p_1 \to e_1 \mid \cdots \mid c_n\ p_n \to e_n]\!]\ \rho} \\
 & = & \underline{\texttt{case}}\ \text{bruijn}[\![e]\!]\ \rho\ \underline{\texttt{of}} \\
 & & c_1 \to \text{bruijn}[\![e_1]\!]\ \langle \rho, p_1 \rangle \mid \cdots \mid c_n \to \text{bruijn}[\![e_n]\!]\ \langle \rho, p_n \rangle \\
\multicolumn{3}{l}{\text{bruijn}[\![\underline{\texttt{let}}\ p_1 = e_1\ \underline{\texttt{in}}\ e]\!]\ \rho} \\
 & = & \underline{\texttt{let}}\ \text{bruijn}[\![e_1]\!]\ \rho\ \underline{\texttt{in}}\ \text{bruijn}[\![e]\!]\ \langle \rho, p_1 \rangle \\
\text{bruijn}[\![\underline{\texttt{rec}}\ p.e]\!]\ \rho & = & \underline{\texttt{rec}}\ \text{bruijn}[\![e]\!]\ \langle \rho, p \rangle
\end{array}
$$

bruijn$[\![f_1\ p_1 = e_1; \ldots ; f_n\ p_n = e_n]\!]$ converts a given program to its de Bruijn variant.

$$
\begin{array}{rcl}
\text{bruijn}[\![\textbf{prg}]\!] & : & \textbf{dbp} \\
\multicolumn{3}{l}{\text{bruijn}[\![f_1\ p_1 = e_1; \ldots ; f_n\ p_n = e_n]\!]} \\
 & = & f_1 = \lambda \text{bruijn}[\![e_1]\!]\ \langle \langle\rangle, p_1 \rangle; \ldots ; f_n = \lambda \text{bruijn}[\![e_n]\!]\ \langle \langle\rangle, p_n \rangle
\end{array}
$$

Table 2.10: Transformation of programs to de Bruijn programs

Chapter 3

Standard Semantics

Semantics is a strange kind of applied mathematics; it seeks profound definitions rather than difficult theorems. The mathematical concepts which are relevant are immediately relevant. Without any long chains of reasoning, the application of such concepts directly reveals regularity in linguistic behaviour, and strengthens and objectifies our intuitions of simplicity and uniformity.

— J.C. Reynolds *(1980)*

In the following we define a denotational semantics for the language introduced in the last chapter. The semantics given is fairly conventional so that readers familiar with the topic may wish to skip this chapter. However, two points are worthy of note. First, the semantics is a strict one as intended by the transformation of Section 2.3. Second, the semantic equations have a rather categorical flavour which is due to the de Bruijn transformation. Both points are put to use in the subsequent chapters.

We essentially employ the notation of Mosses [116] which is introduced and explained in Appendix A. For the mathematical background of the denotational semantic approach we refer to the same appendix.

Plan of the chapter Section 3.1 introduces the semantic domains which are required in the semantic definition. The semantic equations which specify the meaning of the different syntactic constructs are given in Section 3.2.

3.1 Semantic domains

The presentation of the semantic domains proceeds in two steps. Firstly, the domains involved are declared thereby introducing meta variables to be used henceforth to denote elements of the respective domains. The domains are then defined by means of recursive domain equations (Section A.7 explains how to solve them). Note that we will follow this pattern throughout this work, for example, in the definition of the non-standard and the approximating semantics.

Table 3.1 summarizes the semantic domains of the language. Note that the term 'domain' stands for algebraic complete semilattice. The domain **Bas** comprises the semantic counterparts of the system defined types. The remaining domains should be self-explanatory.

The domains are defined via domain equations shown in Table 3.2. Recipes are modeled by lifted values: If e denotes the value v, `freeze` e accordingly denotes **up** v. Lifting captures

meta variable		domain	comment
		Bas	primitive values
v	$\in$	**Val**	values
φ	$\in$	**Fun**	functions
ρ	$\in$	**ValEnv**	value environments
ϕ	$\in$	**FunEnv**	function environments

Table 3.1: Semantic domains

domain equation			comment
Val	$=$	**Bas**	primitive values
	$\oplus$	$\mathbf{Val}_\perp$	lifted values
	$\oplus$	$\mathbf{Val} \oplus \cdots \oplus \mathbf{Val}$	constructed values
	$\oplus$	$\mathbf{Val} \otimes \cdots \otimes \mathbf{Val}$	products
Fun	$=$	$\mathbf{Val}_\perp \otimes \cdots \otimes \mathbf{Val}_\perp \circ\!\!\rightarrow \mathbf{Val}$	functions
ValEnv	$=$	$\mathbf{1}_\perp$	empty environment
	$\oplus$	$\mathbf{ValEnv} \otimes (\mathbf{Val}_\perp \otimes \cdots \otimes \mathbf{Val}_\perp)$	environment layer
FunEnv	$=$	$\mathbf{fun} \rightarrow \mathbf{Fun}$	function environments

Table 3.2: Domain equations

exactly the behaviour of `freeze`, since `freeze` does not alter the value of its argument in any respect apart from avoiding non-termination.

Note that we use coalesced sums, smash products, and strict continuous functions (instead of separated sums, products, and continuous functions) to model data types, sequences, and user-defined functions. This is due to the transformation yielding programs which are to be interpreted in a strict setting. The rules in Table 2.6 basically employ the following domain isomorphisms

$$\begin{array}{rcl} (D \times E)_\perp & \cong & D_\perp \otimes E_\perp, \\ D + E & \cong & D_\perp \oplus E_\perp, \\ D \rightarrow E & \cong & D_\perp \circ\!\!\rightarrow E, \end{array}$$

which show that one may dispense with domain operators $\times$, $+$, and $\rightarrow$ in the presence of their strict counterparts and the lift $(\cdot)_\perp$.

Technical remark: The notation $\mathbf{Val} \oplus \cdots \oplus \mathbf{Val}$ employed in the domain equations serves as a shorthand for $\mathbf{Val}^+$ which is defined by the domain equation $D^+ \cong D \oplus D^+$. Accordingly, $\mathbf{Val} \otimes \cdots \otimes \mathbf{Val}$ stands for $\mathbf{Val}^*$ given by $D^* \cong \mathbf{1}_\perp \oplus D \otimes D^*$.

3.2 Semantic equations

The interpretation of types is given in Table 3.3. Types denote subdomains of **Val**. Since type expressions may contain type variables, their interpretation is given relative to a type environment mapping type variables to domains. Primitive types are interpreted in the obvious way:

The interpretation of the type σ with respect to the type environment ν is given by $\mathcal{T}[\![\sigma]\!]\,\nu$.

$$\begin{array}{rcl}
\mathcal{T}[\![\alpha]\!]\,\nu & = & \nu(\alpha) \\
\mathcal{T}[\![\beta]\!]\,\nu & = & \nu(\beta) \\
\mathcal{T}[\![\varsigma]\!]\,\nu & = & D_\varsigma \\
\mathcal{T}[\![\sigma?]\!]\,\nu & = & (\mathcal{T}[\![\sigma]\!]\,\nu)_\perp \\
\mathcal{T}[\![c_1\sigma_1 \mid \cdots \mid c_n\sigma_n]\!]\,\nu & = & \mathcal{T}[\![\sigma_1]\!]\,\nu \oplus \cdots \oplus \mathcal{T}[\![\sigma_n]\!]\,\nu \\
\mathcal{T}[\![\sigma_1 \ldots \sigma_n]\!]\,\nu & = & \mathcal{T}[\![\sigma_1]\!]\,\nu \otimes \cdots \otimes \mathcal{T}[\![\sigma_n]\!]\,\nu \\
\mathcal{T}[\![\mu\beta.\sigma]\!]\,\nu & = & \mathsf{lfp}(\boldsymbol{\lambda}\delta.\mathcal{T}[\![\sigma]\!](\nu[\beta \mapsto \delta]))
\end{array}$$

Table 3.3: The semantic function $\mathcal{T}$.

$D_\varepsilon = \mathbf{1}_\perp$, $D_{\mathtt{num}} = \mathbb{Z}_\perp$, and $D_{\mathtt{char}} = \mathrm{ASCII}_\perp$. [The meaning of recursive types is somewhat informal, a formal treatment is given in Section A.7].

The semantic functions of the language are displayed in Table 3.4. First-order languages distinguish between ordinary values such as numbers, lists etc and functional values. This is reflected in both syntax and semantics. Variables and user-defined functions belong to different syntactic categories. Accordingly the semantic function $\mathcal{E}$ takes a function and a value environment. While the function environment does not change during the evaluation of an expression, the value environment is dynamically extended as local bindings are introduced. Note that user-defined functions and expressions are interpreted by *strict* functions.

Function environments are modeled as usual by finite mappings. Due to the de Bruijn transformation the interpretation of a value or rather run-time environment is more concrete: It essentially amounts to a list of tuples. Hence an access path $k.i$ corresponds to a composition of projection functions.The k-th entry in a run-time environment is accessed via acc_k : **ValEnv** $\circ\!\!\rightarrow$ **Val** defined by

$$\begin{array}{rcl}
\mathsf{acc}_0 & = & \mathsf{on}_2, \\
\mathsf{acc}_{k+1} & = & \mathsf{acc}_k \circ \mathsf{on}_1.
\end{array}$$

Since a value environment amounts to an ordinary value, expressions respectively their denotations boil down to simple first-order functions. This allows us to treat both user-defined functions and expressions in a uniform manner.

The operators `freeze` and `unfreeze` correspond to lifting and unlifting. The meaning of system-defined functions is fixed in advance; $[\![s]\!]$ denotes the strict, *doubly* lifted variant of s, for instance:

$$\begin{array}{rcll}
[\![\texttt{+}]\!] & : & (\mathbb{Z}_\perp)_\perp \otimes (\mathbb{Z}_\perp)_\perp \circ\!\!\rightarrow \mathbb{Z}_\perp & \\
[\![\texttt{+}]\!](\perp) & = & \perp & \\
[\![\texttt{+}]\!](\mathsf{up}\ v_1, \mathsf{up}\ v_2) & = & \perp & \mathsf{if}\ v_1 = \perp \vee v_2 = \perp \\
 & = & \mathsf{down}\, v_1 + \mathsf{down}\, v_2 & \mathsf{otherwise}
\end{array}$$

The semantic counterpart of <u>`case`</u> is the strict case analysis $[c_1\colon \varphi_1, \ldots, c_n\colon \varphi_n]$ mapping $\mathsf{in}_c\, v$ to $\varphi_i(v)$ if $c_i = c$ and to $\perp$ otherwise. It should be noted that we identify run-time errors such as ‘missing case’ with non-termination. Hence, `head []` evaluates to bottom. This identification is even necessary. Otherwise, the addition would not be strict in both arguments since either `(1/0) +` $\perp \Rightarrow$ *error* or $\perp$ `+ (1/0)` $\Rightarrow$ *error*.

$\mathcal{E}[\![e]\!]\ \phi\ \rho$ denotes the value of e with respect to the function environment ϕ and the value environment ρ.

$$
\begin{aligned}
\mathcal{E}[\![\mathbf{dbe}]\!] &: \mathbf{FunEnv} \to \mathbf{ValEnv} \circ\!\!\to \mathbf{Val} \\
\mathcal{E}[\![k.i]\!]\ \phi\ \rho &= \mathsf{on}_i(\mathsf{acc}_k\ \rho) \\
\mathcal{E}[\![\texttt{freeze}\ e]\!]\ \phi\ \rho &= \mathsf{up}(\mathcal{E}[\![e]\!]\ \phi\ \rho) \\
\mathcal{E}[\![\texttt{unfreeze}\ e]\!]\ \phi\ \rho &= \mathsf{down}(\mathcal{E}[\![e]\!]\ \phi\ \rho) \\
\mathcal{E}[\![s\ e]\!]\ \phi\ \rho &= [\![s]\!]\ (\mathcal{E}[\![e]\!]\ \phi\ \rho) \\
\mathcal{E}[\![c\ e]\!]\ \phi\ \rho &= \mathsf{in}_c\ (\mathcal{E}[\![e]\!]\ \phi\ \rho) \\
\mathcal{E}[\![f\ e]\!]\ \phi\ \rho &= \phi(f)\ (\mathcal{E}[\![e]\!]\ \phi\ \rho) \\
\mathcal{E}[\![e_1 \ldots e_n]\!]\ \phi\ \rho &= \langle \mathcal{E}[\![e_1]\!]\ \phi\ \rho, \ldots, \mathcal{E}[\![e_n]\!]\ \phi\ \rho \rangle \\
\mathcal{E}[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1 \mid \cdots \mid c_n \to e_n]\!]\ \phi\ \rho & \\
&= [c_1 : \boldsymbol{\lambda}^{\circ} v.\mathcal{E}[\![e_1]\!]\ \phi\ \langle \rho, v \rangle, \ldots, c_n : \boldsymbol{\lambda}^{\circ} v.\mathcal{E}[\![e_n]\!]\ \phi\ \langle \rho, v \rangle] \\
&\quad\ (\mathcal{E}[\![e]\!]\ \phi\ \rho) \\
\mathcal{E}[\![\underline{\texttt{let}}\ e_1\ \underline{\texttt{in}}\ e]\!]\ \phi\ \rho &= \mathcal{E}[\![e]\!]\ \phi\ \langle \rho, \mathcal{E}[\![e_1]\!]\ \phi\ \rho \rangle \\
\mathcal{E}[\![\underline{\texttt{rec}}\ e]\!]\ \phi\ \rho &= \mathsf{lfp}_{v_0}(\boldsymbol{\lambda}^{\circ} v.\mathcal{E}[\![e]\!]\ \phi\ \langle \rho, v \rangle) \\
&\quad\ \mathsf{where}\ v_0 = (\mathsf{up}\perp, \ldots, \mathsf{up}\perp)
\end{aligned}
$$

$\mathcal{P}[\![f_1 = \lambda e_1; \ldots ; f_n = \lambda e_n]\!]$ specifies the meaning of the given program.

$$
\begin{aligned}
\mathcal{P}[\![\mathbf{dbp}]\!] &: \mathbf{FunEnv} \\
\mathcal{P}[\![f_1 = \lambda e_1; \ldots ; f_n = \lambda e_n]\!] & \\
&= \mathsf{lfp}(\boldsymbol{\lambda}\phi.\varnothing[f_1 \mapsto \boldsymbol{\lambda} v.\mathcal{E}[\![e_1]\!]\ \phi\ \langle \langle\rangle, v \rangle] \cdots \\
&\qquad [f_n \mapsto \boldsymbol{\lambda} v.\mathcal{E}[\![e_n]\!]\ \phi\ \langle \langle\rangle, v \rangle])
\end{aligned}
$$

Table 3.4: The semantic functions $\mathcal{E}$ and $\mathcal{P}$.

A local value definition simply extends the current value environment. Both the interpretation of recursive expressions and user-defined functions involve fixpoints. However, there is a subtle difference between them. While the latter amounts to an ordinary fixpoint iteration, the former takes the least fixpoint above $v_0 = (\mathsf{up}\perp, \ldots, \mathsf{up}\perp)$ in order to avoid the trivial fixpoint $\perp$. [Technical remark: $\mathsf{lfp}_{v_0}(\varphi) := \bigsqcup_{n \geqslant 0} \varphi^n(v_0)$ is only well-defined if v_0 is a pre-fixpoint of φ. This is granted, since strictness of $\mathcal{E}$ implies that the value environment ρ is proper.]

3.3 Bibliographic notes

Kubiak *et. al.* [105] were the first to use lifts in the standard semantics of a non-strict language (at least in a paper concerned with strictness analysis). The importance of lifts was already recognized in the first paper on projection-based strictness analysis by Wadler and Hughes [150]. However, the argument given in *loc. cit.* was not very convincing, so the construction seemed to be not more than just a technical trick. [Even the notation was somewhat unusual, the lifting operator being symbolized by a lightning bolt $\lightning$.] The connection between the semantic model and the internal workings of a machine was established by Hughes and Launchbury [85].

Chapter 4

Projections

What is hell?
Hell is oneself,
Hell is alone, the other figures in it
Merely projections. There is nothing to escape from
And nothing to escape to. One is always alone.
— T.S. Eliot, *The Cocktail Party (1950) act 1, sc. 3*

We have seen in the introduction that the evaluation of an expression depends crucially on the context in which it is situated. The context `sum` $\cdot$, for example, places a stronger demand on an expression than the context `length` $\cdot$. The evaluation of an expression may be considered as driven by these demands. For that reason lazy evaluation is often coined 'demand-driven' evaluation.

There have been several attempts both informal and formal to explain the notion of context. An early paper [79] is based on the continuation semantics (as opposed to the direct semantics given in Chapter 3) which explicitly models the context of an expression by means of a continuation function[1]. This work forms, in fact, the basis for the seminal paper on projection-based strictness analysis by P. Wadler and J. Hughes [150].

Projections abstract from continuations like `sum` $\cdot$ or `product` $\cdot$ in that they only model the 'degree of evaluation' caused by the continuation. This is, however, exactly what we have in mind. With respect to strictness properties `sum` $\cdot$ and `product` $\cdot$ may be considered as equivalent since both fully evaluate their arguments. We do not really care that `sum` adds the elements of its argument list while `product` multiplies them. Projections capture nicely the idea of pre-evaluating an argument. If an argument is required head and tail strict the projection 'HT' may be applied to it prior to the function call. Thus projections allow to extend the "The Theory and Practice of Transforming Call-By-Need into Call-By-Value" to non-flat domains in a very natural way.

This chapter develops a theory of projections as required by the subsequent chapters. Note that basic material is to be found in Section A.7. The results fall largely in two categories: representation of projections and algebraic properties of operations on projections.

One attractive feature of the theory is that it can be easily visualized. The reader will notice that we make heavy use of diagrams and diagrammatic arguments in this and the next chapter. This approach was, in fact, inspired by the excellent textbook [45] on lattices and order.

[1] Loosely speaking a continuation function specifies what 'the rest of a program' will do with the value of a particular expression.

Plan of the chapter Section 4.1 is concerned with basic mathematical properties of projections. Like domains complex projections may be constructed from simple ones. Each domain constructor $(\cdot)_\perp$, $\oplus$, and $\otimes$ gives rise to several characteristic operations on projections, which are introduced in Section 4.2, 4.3, and 4.4 respectively. Typically, operations come in pairs: a set of operations to synthesize new projections and a set of operations to analyse given projections. Section 4.5 employs these operations to show that classical strictness properties such as bistrictness can easily be expressed in terms of projections. If a variable appears more than once in an expression different demands must be combined in an appropriate way. Section 4.6 deals with the operations necessary. We conclude with an aside on smash projections in Section 4.7.

4.1 Properties of projections

Demands like 'H' or 'T' specify a certain degree of evaluation. The demand 'T', for example, specifies that the spine of its argument list is definitely evaluated, while the elements of the list are probably not evaluated. Semantically the process of evaluation may be characterized by the following two properties.

1. Evaluation of an expression does not change its value apart from introducing non-termination. Thus the denotation of an evaluated expression is weaker than the denotation of the unevaluated expression with respect to Scott's information ordering.

2. An evaluator reduces an expression to a normalform. Applying the evaluator twice yields the same normalform.

These properties are formally captured by the notion of projection. Projections were first introduced by Scott, see [64] for an overview, in order to solve recursive domain equations (cf Section A.7). Later on Wadler [150] discovered that the notion of context could be explained by projections in a very satisfactory way.[2]

4.1 Definition *Let D be a domain. A continuous function $\pi : D \to D$ is called a* projection *iff it is idempotent ($\pi \circ \pi = \pi$) and reductive ($\pi \sqsubseteq \boldsymbol{\lambda} v.v$). A projection π is termed* finitary *iff $\mathbf{im}\,\pi := \{\, \pi\, v \mid v \in D \,\}$ is a domain. The set of all finitary projections on D is denoted $\|D\|$.*

Note that $\pi \sqsubseteq \boldsymbol{\lambda} v.v$ implies $\pi\ \bot = \bot$, that is, projections are strict. We will make constant use of this fact in the proofs following. Furthermore note that the property of idempotence is a bit stronger than necessary: $\pi \circ \pi = \pi$ may be replaced by $\pi \circ \pi \sqsupseteq \pi$, since $\pi \sqsubseteq \boldsymbol{\lambda} v.v$ already implies $\pi \circ \pi \sqsubseteq \pi$.

Subsequently we will only consider finitary projections. The reason for this restriction is rather empirical: The domains which we employ in the standard semantics only accommodate finitary projections (cf Lemma A.25). Furthermore we profit from the results of domain theory where it is very natural to impose this condition.

If a strict function, say, φ is tail strict, the context or rather the projection 'T' may be applied to its argument, say, v without altering the value of the function call: $\varphi(\mathrm{T}\, v) = \varphi(v)$. If $\mathrm{T}\, v$ yields $\bot$, we may conclude that $\varphi(v)$ is undefined as well. Thus projections specify the amount of information which is *necessary* for the function call to work. A value v is termed *acceptable* to a projection π iff $\pi(v) \neq \bot$. The value $v = \mathbf{up}\,\mathbf{f} : \mathbf{up}\,\bot : \mathbf{up}\,\bot$ is acceptable

[2] The paper is written by P. Wadler and J. Hughes. The idea to model contexts by projections, however, is attributed to P. Wadler.

to 'H' but unacceptable to 'T'. Note that v being acceptable does not imply that v provides *sufficient* information; consider, for example, `and` v which is undefined though v is acceptable to 'H'. In other words, strictness establishes a lower bound on the definedness of values.

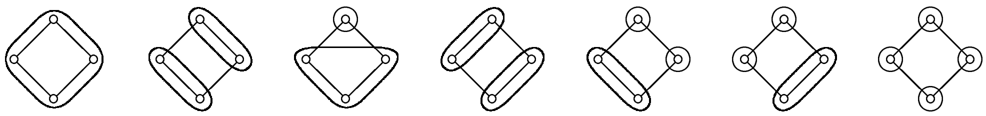
Examples for projections are $\perp\!\!\!\perp := \boldsymbol{\lambda} v.\bot$ (pronounce 'bottom') and $\top\!\!\!\top := \boldsymbol{\lambda} v.v$ (pronounce 'identity'). If the underlying domain D is finite, projections can be represented using the Hasse diagram of D by surrounding points with the same image. On the two-point domain $\mathbf{2} \cong \mathbf{1}_\bot$ the only projections are, in fact, bottom and identity. They may be depicted as follows.

$\perp\!\!\!\perp$ $\top\!\!\!\top$

The three-point domain $\mathbf{3} \cong \mathbf{2}_\bot$ accommodates four different projections (the notation is introduced only in Section 4.2).

$\perp\!\!\!\perp_{\obar}$ $\perp\!\!\!\perp_{\bot}$ $\top\!\!\!\top_{\obar}$ $\top\!\!\!\top_{\bot}$

As a final example consider the Boolean lattice $\mathbf{2} \times \mathbf{2}$. On $\mathbf{2} \times \mathbf{2}$ there are thirty-six monotonic functions, seven of which satisfy the additional properties of projections.

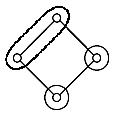
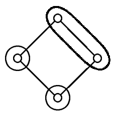


Monotonicity and idempotence of projections imply that each block is convex, since $\pi(v) \sqsubseteq w \sqsubseteq v$ implies $\pi(v) = \pi(w)$. Equally obvious, each block has a least element, which is the common image of the elements contained in the block. The diagrammatical representation ensures that the corresponding function is idempotent and that it approximates the identity, thus excluding functions like f or g.

$\pi \sqsubseteq \boldsymbol{\lambda} v.v$ ↯   $\pi \circ \pi = \pi$ ↯

f g

By contrast monotonicity is not granted, since this property relates elements of different blocks in particular.

 monotonicity ↯ 

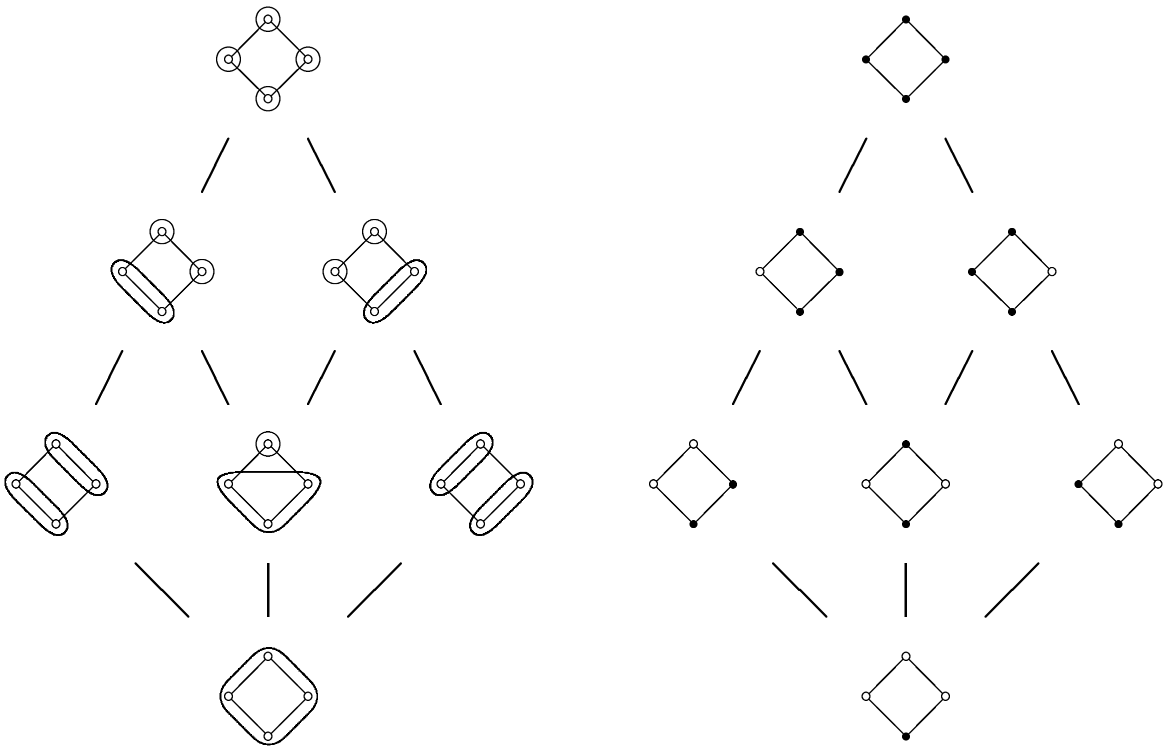
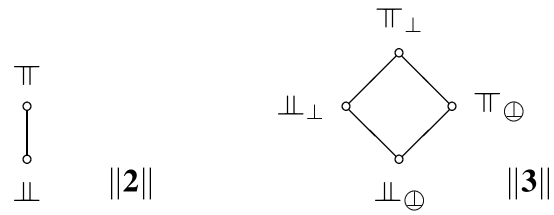


Figure 4.1: Projections on $\mathbf{2} \times \mathbf{2}$ and their images

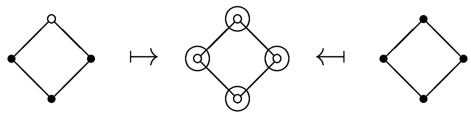
Projections on D inherit the pointwise ordering of the function space $D \to D$. The projections on **2** and **3**, for example, have the following domain ordering.



Note that the pointwise ordering of projections is reflected by the subset ordering of their images: $⫫_{\oplus}$ has one, $⫫_{\perp}$ and $⊤⊤_{\oplus}$ have two and $⊤⊤_{\perp}$ has three fixpoints. The lattice of projections on $\mathbf{2} \times \mathbf{2}$ displayed in Figure 4.1 further substantiates this observation. It is evident that each projection on $\mathbf{2} \times \mathbf{2}$ is uniquely determined by its image. Generally, we have the following: A finitary projection is determined by the *finite* elements of its image. Assume that a set of finite elements $N \subseteq F(D)$ is given, then π_N with

$$\pi_N(v) \quad = \quad \bigsqcup N \cap {\downarrow} v$$

is the least finitary projection such that $N \subseteq F(\textbf{im}\,\pi)$. Note that π_N is well-defined, since $N \cap {\downarrow} v$ is bounded. However, the map $N \mapsto \pi_N$ is not one-to-one as the following example shows.



There is no projection whose image coincides with the set on the left hand side, since the corresponding functions are not monotonic (see above). The question naturally arises as to whether there is an easy way of pinpointing sets of finite elements which do not correspond to finitary projections. The definition of finite elements provides a clue. An element $v \in D$ is called finite iff for every *directed* set $S \subseteq D$, $v \sqsubseteq \bigsqcup S$ implies $v \sqsubseteq s$, for some $s \in S$. Now, if $v \in F(\mathbf{im}\,\pi_N)$ and $N \cap {\downarrow}v$ is directed (not only bounded), we may conclude that $v \in N$. This motivates the following definition.

4.2 Definition *Let P be an ordered set and let $N \subseteq P$. Then N is said to be* normal *in P (notation: $N \lhd P$) iff the set $N \cap {\downarrow}v$ is directed for every $v \in P$. The set of all normal substructures of P is denoted by $\mathfrak{N}(P)$.*

Directed sets may not be empty by definition. Hence $\bot$ is contained in each normal set. The only non-normal set containing $\bot$ is in fact $\{00, 01, 10\}$. Thus there are $2^3 - 1 = 7$ normal sets in $\wp(\mathbf{2} \times \mathbf{2})$. Now, one can show that $\pi \mapsto F(\mathbf{im}\,\pi)$ and $N \mapsto \pi_N$ establish a one-to-one correspondence between $\|D\|$ and $\mathfrak{N}(F(D))$ (cf Theorem A.16). Using this characterization the following is easy to prove.

4.3 Fact *Let D be a domain. The set $\|D\|$ of finitary projections on D forms an algebraic lattice.*

A proof which is to be found in Section A.7 (cf Theorem A.18) amounts to showing that $\mathfrak{N}(F(D))$ is a topped algebraic $\bigcap$-structure.

Since we work in an abstract interpretation setting Fact 4.3 is, of course, very attractive. In the current and in the following chapters we will make intensive use of this fact. The algebraicity, for example, provides the key for representing projections.

Each topped $\bigcap$-structure on a set X induces a closure operator on $\wp(X)$. The following lemma gives a constructive characterization of the closure operator associated with the set of normal substructures of $F(D)$. It is based on the observation that normal subsets are closed under finite joins: Let $F \Subset F(\mathbf{im}\,\pi)$ be a finite subset. Since π is reductive we have $\pi(\bigsqcup F) \sqsubseteq \bigsqcup F$. On the other hand monotonicity implies $\pi(\bigsqcup F) \sqsupseteq \bigsqcup \pi(F) = \bigsqcup F$. Returning to our running example, projections on $\mathbf{2} \times \mathbf{2}$, we see that $\{00, 01, 10\}$ is the only set which is not closed under joins.

4.4 Lemma *Let D be a domain. Then $\mathbf{c} : \wp(F(D)) \to \wp(F(D))$ with*

$$\mathbf{c}(M) \quad = \quad \{\textstyle\bigsqcup F \mid F \Subset M \text{ and } F \text{ bounded}\}$$

is the closure operator associated with the $\bigcap$-structure $\langle \mathfrak{N}(F(D)); \subseteq \rangle$.

Proof We first show that $\mathbf{c}$ is indeed a closure operator. Clearly, $\mathbf{c}$ is monotone and $M \subseteq \mathbf{c}(M)$. To see that $\mathbf{c}$ is idempotent, let $v \in \mathbf{c}(\mathbf{c}(M))$. Thus there is a finite family $\{F_i\}_{i \in I}$ of sets $F_i \Subset M$, for all $i \in I$, such that $v = \bigsqcup\{\bigsqcup F_i\}_{i \in I}$. Furthermore each F_i is bounded and $\{\bigsqcup F_i\}_{i \in I}$ is bounded. Hence, $\bigcup\{F_i\}_{i \in I}$ is bounded as well and we have

$$v = \textstyle\bigsqcup\{\bigsqcup F_i\}_{i \in I} = \bigsqcup(\bigcup\{F_i\}_{i \in I}) \in \mathbf{c}(M).$$

It remains to show that $\mathbf{c}(M)$ is normal and that $\mathbf{c}(M)$ is the least normal set containing M. Assume that $v \in F(D)$ and $F \Subset \mathbf{c}(M) \cap {\downarrow}v$. Idempotence of $\mathbf{c}$ implies that $\bigsqcup F \in \mathbf{c}(M)$. Consequently, $\mathbf{c}(M)$ is normal in $F(D)$. Finally, let $N \lhd F(D)$ with $M \subseteq N$ and let $v \in \mathbf{c}(M)$. Thus there is a subset $F \Subset M$ such that $\bigsqcup F$ exists and $v = \bigsqcup F$. Since N is normal, Lemma A.17 implies $v = \bigsqcup F \in N$. ■

Less concretely we could have defined the closure operator by $\mathbf{c}(M) = F(\mathbf{im}\,\pi_M)$. [Since $M \subseteq F(\mathbf{im}\,\pi_M)$ and $F(\mathbf{im}\,\pi_M)$ is normal we know that $\mathbf{c}(M) \subseteq F(\mathbf{im}\,\pi_M)$. On the other hand $M \subseteq \mathbf{c}(M)$ implies $F(\mathbf{im}\,\pi_M) \subseteq F(\mathbf{im}\,\pi_{\mathbf{c}(M)}) = \mathbf{c}(M)$.]

The following lemma shows that the join is formed pointwise in $\|D\|$.

4.5 Lemma *Let D be a domain and let $\Pi \subseteq \|D\|$. Then*

$$(\bigsqcup \Pi)(v) \quad = \quad \bigsqcup\{\, \pi(v) \mid \pi \in \Pi \,\}.$$

Proof We basically employ the correspondence between finitary projections and normal substructures. Let $\varphi(\pi) = F(\mathbf{im}\,\pi)$. Since φ is an order-isomorphism, we have $\varphi(\bigsqcup \Pi) = \bigsqcup \varphi(\Pi) = \mathbf{c}(\bigcup \varphi(\Pi))$, for $\Pi \subseteq \|D\|$.

$$\begin{aligned}
(\bigsqcup \Pi)(v) &= \bigsqcup \varphi(\bigsqcup \Pi) \cap {\downarrow} v && \pi = \pi_{F(\mathbf{im}\,\pi)} \\
&= \bigsqcup \mathbf{c}(\bigcup \varphi(\Pi)) \cap {\downarrow} v && \varphi(\bigsqcup \Pi) = \mathbf{c}(\bigcup \varphi(\Pi)) \\
&= \bigsqcup (\bigcup \varphi(\Pi)) \cap {\downarrow} v && \pi_M = \pi_{\mathbf{c}(M)} \\
&= \bigsqcup \bigcup \{\, \varphi(\pi) \cap {\downarrow} v \mid \pi \in \Pi \,\} && \\
&= \bigsqcup \{\, \bigsqcup \varphi(\pi) \cap {\downarrow} v \mid \pi \in \Pi \,\} && \\
&= \bigsqcup \{\, \pi(v) \mid \pi \in \Pi \,\} && \pi = \pi_{F(\mathbf{im}\,\pi)} \;\blacksquare
\end{aligned}$$

Whereas the supremum on $\|D\|$ coincides with the one on $D \to D$ this is not the case with the infimum, for example, $\perp\!\!\!\perp_{\perp} \sqcap_{\mathbf{3}\to\mathbf{3}} \top\!\!\!\top_{\oplus} = g$ but $\perp\!\!\!\perp_{\perp} \sqcap_{\|\mathbf{3}\|} \top\!\!\!\top_{\oplus} = \perp\!\!\!\perp_{\oplus}$.

Now, let us turn to the representation of projection lattices. As $\|D\|$ is an algebraic lattice, a projection π is completely characterized by the set $F(\|D\|) \cap {\downarrow}\pi$ of its finite approximations. It remains to identify the finite elements of $\|D\|$. We know that an element N of a topped algebraic $\bigcap$-structure with closure operator C is finite iff $N = C(M)$ for a finite M. Because the closure operator $\mathbf{c}$ preserves finiteness of sets, a finitary projection is finite iff $F(\mathbf{im}\,\pi)$ is finite.

The following theorem shows that we can even confine ourselves to projections having exactly one non-trivial fixpoint—this is analogous to the lattice of sets where every set is equal to a union of singleton sets. Let $F^\circ(D) = F(D) \setminus \{\perp\}$, then it is easy to see that the two-element sets $\{\perp, a\}$ with $a \in F^\circ(D)$ are normal. Order-theoretically, the projections $\pi_{\{\perp,a\}}$ with $a \in F^\circ(D)$ are exactly the elements in $\|D\|$ which cover the bottom projection, that is, they are the atoms of the projection lattice.

4.6 Theorem *Let D be a domain and let $[a] := \pi_{\{\perp,a\}}$, for $a \in F^\circ(D)$. Then, for each $\pi \in \|D\|$,*

$$\pi \quad = \quad \bigsqcup\{\, [a] \mid a \in F^\circ(\mathbf{im}\,\pi) \,\}.$$

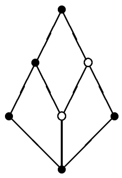
Proof It suffices to prove that $\pi_N = \bigsqcup\{\, [a] \mid a \in N \,\}$ for $N \subseteq F^\circ(D)$, since $\pi = \pi_{F^\circ(\mathbf{im}\,\pi)}$. Let $\varphi(\pi) = F(\mathbf{im}\,\pi)$, then

$$\begin{aligned}
\varphi(\pi_N) &= \mathbf{c}(N) && \text{remark after Lemma 4.4} \\
&= \mathbf{c}(\bigcup\{\, \{\perp, a\} \mid a \in N \,\}) && \mathbf{c}(N \cup \{\perp\}) = \mathbf{c}(N) \\
&= \mathbf{c}(\bigcup\{\, \varphi([a]) \mid a \in N \,\}) && \{\perp, a\} \text{ is normal} \\
&= \varphi(\bigsqcup\{\, [a] \mid a \in N \,\}) && \varphi(\bigsqcup \Pi) = \mathbf{c}(\bigcup \varphi(\Pi))
\end{aligned}$$

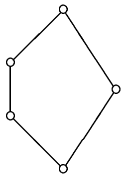
Consequently, $\pi = \bigsqcup\{\, [a] \mid a \in F^\circ(\mathbf{im}\,\pi) \,\}$ by Theorem A.16.

Note that the projection $[a]$ maps the elements of $\uparrow a$ to a and the remaining elements to $\perp$.

On the negative side lattices of projections lack additional algebraic structure such as modularity or distributivity as the lattice $\|\mathbf{2} \times \mathbf{2}\|$ shows.



$\|\mathbf{2} \times \mathbf{2}\|$



$\mathbf{N}_5$

The shaded elements form a sublattice which is isomorphic to $\mathbf{N}_5$. A famous result attributed to R. Dedekind implies that $\|\mathbf{2} \times \mathbf{2}\|$ is not modular (and so also not distributive).

4.2 Projections on lifted domains

The purpose of this and the next two sections is to develop a 'compositional representation theory': Knowing how to represent projections on $D_1, \ldots, D_n$ we seek for a representation of projections on $\Phi(D_1, \ldots, D_n)$ where Φ is one of the domain constructors $(\cdot)_\perp$, $\oplus$, and $\otimes$.

Note that we exclusively deal with *finitary* projections in the sequel. By Lemma A.25 we know that first-order domains only accommodate finitary projections. For that reason we refrain from giving detailed proofs that the operations to be introduced indeed preserve this property. Furthermore, we will often be sloppy and omit the qualifier 'finitary'.

The lift $D_\perp$ is used in the standard semantics to model unevaluated expressions. The domain $D_\perp$ essentially extends D by the element $\mathsf{up}\,\perp$ representing a recipe which does not terminate if forced. With respect to the lattice of projections there are only two possibilities of extending a projection on D to a projection on $D_\perp$: Either $\mathsf{up}\,\perp$ is added to the image of the given projection or it is not. The former choice corresponds to call-by-need (termed 'lazy lift' below) while the latter corresponds to call-by-value (termed 'strict lift').

4.7 Definition *Let π be a projection on D. The projections $\pi_{\textcircled{\scriptsize$\perp$}}$ (pronounce 'strict lift') and $\pi_\perp$ (pronounce 'lazy lift') on $D_\perp$ are defined as follows.*

$$\pi_{\textcircled{\scriptsize$\perp$}} \;=\; \mathsf{strict\ up} \circ \pi \circ \mathsf{down} \qquad\qquad \pi_\perp \;=\; \mathsf{strict}(\mathsf{up} \circ \pi \circ \mathsf{down})$$

Conversely, let π be a projection on $D_\perp$. The projection $\pi{\downarrow}$ on D is defined as follows.

$$\pi{\downarrow} \;=\; \mathsf{down} \circ \pi \circ \mathsf{up}$$

Finally, let

$$\begin{aligned} \Lambda(\pi) &= \textcircled{\scriptsize$\perp$} \quad \mathsf{if}\ \pi(\mathsf{up}\ \perp) = \perp \\ &= \perp \quad \mathsf{otherwise}. \end{aligned}$$

The reader may convince herself that $\pi_{\textcircled{\scriptsize$\perp$}}$, $\pi_\perp$, and $\pi{\downarrow}$ are indeed projections. The following explication of lifts is due to Kubiak *et. al.* [105]. A demand of the form $\pi_{\textcircled{\scriptsize$\perp$}}$ may be interpreted as follows, "the value is definitely required, and what is more, π's worth of it will be needed". Conversely, a demand of the form $\pi_\perp$ means, "the value may or may not be required, but if it is then π's worth of it will be needed". [Categorically speaking, the functions $(\cdot)_{\textcircled{\scriptsize$\perp$}}$ and $(\cdot)_\perp$ are the morphism parts of the covariant functors $(\cdot)_{\textcircled{\scriptsize$\perp$}}, (\cdot)_\perp : \mathcal{CPO} \to \mathcal{CPO}_\perp$, where $\mathcal{CPO}_\perp$ is the subcategory of strict continuous functions. The functor $(\cdot)_\perp$ preserves identity while $(\cdot)_{\textcircled{\scriptsize$\perp$}}$ does not.]

The operations $\downarrow$ and Λ allow to factor a projection on $D_\perp$ into its lift and a projection on D. The next lemma shows that strict and lazy lift indeed suffice to represent projections on lifted domains.

4.8 Lemma *Let $\bar{\pi}$ be a projection on $D_\perp$, π a projection on D and $\ell \in \{\oplus, \perp\}$. The following equalities hold*

1. $\bar{\pi} = (\bar{\pi}\downarrow)_{\Lambda(\bar{\pi})}$,
2. $(\pi_\ell)\downarrow = \pi$,
3. $\Lambda(\pi_\ell) = \ell$.

Furthermore, we have $\|D_\perp\| \cong \|D\| \times \mathbf{2}$.

Proof Since projections are strict, it suffices to show the proposition for proper values. Ad 1) Case $\bar{\pi}(\mathbf{up}\ \perp) = \perp$: We show first that this implies $(\forall v)\, \bar{\pi}(\mathbf{up}\ v) \neq \mathbf{up}\ \perp$. Proof by contradiction: Assume that $(\exists v)\, \bar{\pi}(\mathbf{up}\ v) = \mathbf{up}\ \perp$.

$$\mathbf{up}\ \perp = \bar{\pi}(\mathbf{up}\ v) = \bar{\pi}(\bar{\pi}(\mathbf{up}\ v)) = \bar{\pi}(\mathbf{up}\ \perp) = \perp \quad \lightning$$

Using $\mathbf{strict\ up}(\mathbf{down}\ v) = v \iff v \neq \mathbf{up}\ \perp$ the desired equation is easily derived.

$$(\bar{\pi}\downarrow)_\oplus \circ \mathbf{up} = \mathbf{strict\ up} \circ \bar{\pi}\downarrow = \mathbf{strict\ up} \circ \mathbf{down} \circ \bar{\pi} \circ \mathbf{up} = \bar{\pi} \circ \mathbf{up}$$

Case $\bar{\pi}(\mathbf{up}\ \perp) = \mathbf{up}\ \perp$: This implies $(\forall v)\, \bar{\pi}(\mathbf{up}\ v) \neq \perp$.

$$\mathbf{up}\,\perp \sqsubseteq \mathbf{up}\, v \implies \bar{\pi}(\mathbf{up}\,\perp) \sqsubseteq \bar{\pi}(\mathbf{up}\, v) \implies \mathbf{up}\,\perp \sqsubseteq \bar{\pi}(\mathbf{up}\, v)$$

Using $\mathbf{up}(\mathbf{down}\, v) = v \iff v \neq \perp$ the proposition follows immediately.

$$(\bar{\pi}\downarrow)_\perp \circ \mathbf{up} = \mathbf{up} \circ \bar{\pi}\downarrow = \mathbf{up} \circ \mathbf{down} \circ \bar{\pi} \circ \mathbf{up} = \bar{\pi} \circ \mathbf{up}$$

Ad 2) Follows from $\mathbf{down} \circ \mathbf{strict\ up} = \mathbf{id} = \mathbf{down} \circ \mathbf{up}$. Ad 3) Follows immediately from the definition.

The isomorphism between $\|D_\perp\|$ and $\|D\| \times \mathbf{2}$ is established by the monotonic maps $\pi \mapsto (\pi\downarrow, \Lambda(\pi))$ and $(\pi, \ell) \mapsto \pi_\ell$. ■

The characterization of $\|D_\perp\|$ implies that lifting doubles the number of projections. Thus there are indeed four projections on $\mathbf{3} \cong \mathbf{2}_\perp$, namely $\perp\!\!\!\perp_\oplus$, $\perp\!\!\!\perp_\perp$, $\top\!\!\!\top_\oplus$, and $\top\!\!\!\top_\perp$.

The following fact is the connection link between the general representation theory (Theorem 4.6) and its specialization to lifts (Lemma 4.8).

4.9 Fact *Let D be a domain and let $x \in F^\circ(D)$, then*

$$\begin{aligned} [\mathbf{up}\,\perp] &= \perp\!\!\!\perp_\perp, \\ [\mathbf{up}\, x] &= [x]_\oplus. \end{aligned}$$

Hence, each projection on a lifted domain $D_\perp$ is either equal to $\pi_\oplus$ or to $\pi_\oplus \sqcup \perp\!\!\!\perp_\perp$, for some $\pi \in \|D\|$.

4.3 Projections on coalesced sums

Projections on sum domains are constructed using 'case analysis' and injection functions.

4.10 Definition *Let π_i be a projection on D_i. The projection $\pi_1 \oplus \cdots \oplus \pi_n$ on $D_1 \oplus \cdots \oplus D_n$ is defined as follows.*

$$\pi_1 \oplus \cdots \oplus \pi_n \quad = \quad [\mathbf{in}_1 \circ \pi_1, \ldots, \mathbf{in}_n \circ \pi_n]$$

Conversely, let π be a projection on $D_1 \oplus \cdots \oplus D_n$. The projection $\pi \downarrow i$ on D_i is defined as follows.

$$\pi \downarrow i \quad = \quad \mathbf{out}_i \circ \pi \circ \mathbf{in}_i$$

[Note that the function $\cdot \oplus \cdots \oplus \cdot$ is the morphism part of the covariant functor $\cdot \oplus \cdots \oplus \cdot : \mathcal{CPO}_\perp^n \to \mathcal{CPO}_\perp$.] The simplest example for coalesced sums is the domain of boolean values $\mathbb{B}_\perp \cong \mathbf{2} \oplus \mathbf{2}$, which gives rise to four different projections.

$\perp\!\!\!\perp \oplus \perp\!\!\!\perp$ $\quad$ $\perp\!\!\!\perp \oplus \top\!\!\!\top$ $\quad$ $\top\!\!\!\top \oplus \perp\!\!\!\perp$ $\quad$ $\top\!\!\!\top \oplus \top\!\!\!\top$

Each of the projections may be defined in terms of $\cdot \oplus \cdot$. The following lemma shows that this is always the case.

4.11 Lemma *Let π be a projection on $D_1 \oplus \cdots \oplus D_n$ and π_i projections on D_i. The following equalities hold*

1. $\pi = \pi \downarrow 1 \oplus \cdots \oplus \pi \downarrow n$,
2. $(\pi_1 \oplus \cdots \oplus \pi_n) \downarrow i = \pi_i$.

Furthermore, we have $\|D_1 \oplus \cdots \oplus D_n\| \cong \|D_1\| \times \cdots \times \|D_n\|$.

Proof Ad 1) Since $\pi(\mathbf{in}_i\, v) \sqsubseteq \mathbf{in}_i\, v$ the equation follows using $\mathbf{is}_i(v) \Longrightarrow \mathbf{in}_i(\mathbf{out}_i\, v) = v$.

$$\begin{aligned}
(\pi \downarrow 1 \oplus \cdots \oplus \pi \downarrow n)(\mathbf{in}_i\, v) &= (\mathbf{in}_i \circ \pi \downarrow i)(v) && \text{def. } \oplus \\
&= (\mathbf{in}_i \circ \mathbf{out}_i \circ \pi \circ \mathbf{in}_i)(v) && \text{def. } \cdot \downarrow i \\
&= \pi(\mathbf{in}_i\, v) && \mathbf{is}_i(\pi(\mathbf{in}_i\, v))
\end{aligned}$$

Ad 2) Using $\mathbf{out}_i(\mathbf{in}_i\, v) = v$ the proposition follows directly.

$$(\pi_1 \oplus \cdots \oplus \pi_n) \downarrow i = \mathbf{out}_i \circ (\pi_1 \oplus \cdots \oplus \pi_n) \circ \mathbf{in}_i = \mathbf{out}_i \circ \mathbf{in}_i \circ \pi_i = \pi_i$$

The monotonic maps $\pi \mapsto (\pi \downarrow 1, \ldots, \pi \downarrow n)$ and $(\pi_1, \ldots, \pi_n) \mapsto \pi_1 \oplus \cdots \oplus \pi_n$ establish a lattice isomorphism between $\|D_1 \oplus \cdots \oplus D_n\|$ and $\|D_1\| \times \cdots \times \|D_n\|$. ■

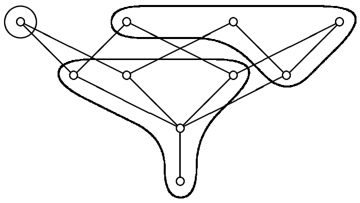
Projections on sums containing bottom projections may be used to characterize functions whose case analysis is incomplete. Consider as a simple example the function `head` with

```
head x = case x of a:w → a,
```

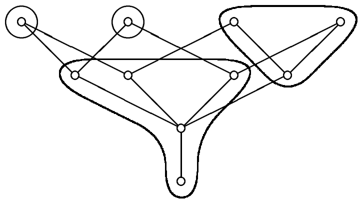
which is undefined on the empty list. The least projection satisfying `head` $=$ `head` $\circ\, \pi$ is $(\perp\!\!\!\perp \oplus \top\!\!\!\top_{\oplus} \otimes \perp\!\!\!\perp_{\perp})_{\oplus}$ (the operator $\otimes$ is introduced only in Definition 4.12).

4.4 Projections on smash products

The attentive reader may have noticed that operations on $\|D_\perp\|$ and $\|D_1 \oplus \cdots \oplus D_n\|$ closely correspond to operations on $D_\perp$ and $D_1 \oplus \cdots \oplus D_n$ respectively. If we try to push the analogy further, we are faced with a problem. Given a projection on $D_1 \otimes \cdots \otimes D_n$ there is no obvious way of factoring it onto projections on the component domains. Consider the projection π_{sor} on $(\mathbb{B}_\perp)_\perp \otimes (\mathbb{B}_\perp)_\perp$ which is pictured on the right.



sequential or



π_{sor}

As the name suggests, π_{sor} characterizes the behaviour of 'sequential or', which is displayed on the left hand side. It is, in fact, the least projection such that $\mathrm{sor} = \mathrm{sor} \circ \pi_{\mathrm{sor}}$. The effect of π_{sor} on one component is not independent of the other component's value: fixing $\mathbf{up\,f}$ in the first component and varying the second component yields $\mathbf{on}_1(\pi_{\mathrm{sor}}(\mathbf{up\,f}, \mathbf{up}\,\perp)) = \perp$ and $\mathbf{on}_1(\pi_{\mathrm{sor}}(\mathbf{up\,f}, \mathbf{up\,f})) = \mathbf{up\,f} = \mathbf{on}_1(\pi_{\mathrm{sor}}(\mathbf{up\,f}, \mathbf{up\,t}))$, respectively. Since our ultimate goal is to derive *necessary* conditions on the definedness of values, we are forced to assume the weakest possible outcome or more generally their least upper bound, $\mathbf{up\,f}$ in the example above.

4.12 Definition *Let π_i be a projection on D_i. The projection $\pi_1 \otimes \cdots \otimes \pi_n$ on $D_1 \otimes \cdots \otimes D_n$ is defined as follows.*

$$\begin{aligned} (\pi_1 \otimes \cdots \otimes \pi_n)(\perp) &= \perp \\ (\pi_1 \otimes \cdots \otimes \pi_n)(v_1, \ldots, v_n) &= \langle \pi_1\, v_1, \ldots, \pi_n\, v_n \rangle \end{aligned}$$

Conversely, let π be a projection on $D_1 \otimes \cdots \otimes D_n$. The projection $\pi \uparrow i$ on D_i is defined as follows.

$$(\pi \uparrow i)(v) = \bigsqcup \{\, \mathbf{on}_i(\pi\, w) \mid \mathbf{on}_i(w) = v \,\}$$

[As to be expected the function $\cdot \otimes \cdots \otimes \cdot$ defines the morphism part of the covariant functor $\cdot \otimes \cdots \otimes \cdot : \mathcal{CPO}_\perp^n \to \mathcal{CPO}_\perp$.] Projections of the form $\pi_1 \otimes \cdots \otimes \pi_n$ work on each component separately and do not combine different components with each other. Clearly, this is too restrictive to represent every possible projection on products.

4.13 Lemma *Let π be a projection on $D_1 \otimes \cdots \otimes D_n$ and let $\pi_i \neq \perp\!\!\!\perp$ be projections on D_i. The following (in-) equalities hold*

1. $\pi \sqsubseteq \pi \uparrow 1 \otimes \cdots \otimes \pi \uparrow n$,

2. $(\pi_1 \otimes \cdots \otimes \pi_n) \uparrow i = \pi_i$.

Proof Ad 1) Case $\pi(v_1, \dots, v_n) = \bot$: Holds trivially. Case $\pi(v_1, \dots, v_n) \neq \bot$: Hence $(\pi \uparrow i)(v_i) \neq \bot$, for all i. Note that $v \sqsubseteq v' \iff (\forall i)\ \mathsf{on}_i\, v \sqsubseteq \mathsf{on}_i\, v'$.

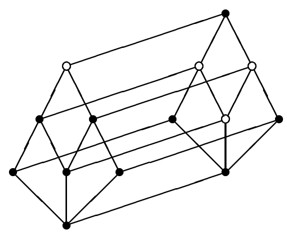
$$
\begin{array}{cll}
 & (\mathsf{on}_i \circ (\pi \uparrow 1 \otimes \cdots \otimes \pi \uparrow n))(v_1, \dots, v_n) & \\
= & \mathsf{on}_i(\langle (\pi \uparrow 1)(v_1), \dots, (\pi \uparrow n)(v_n) \rangle) & \text{def. } \otimes \\
= & (\pi \uparrow i)(v_i) & (\forall i)\ (\pi \uparrow i)(v_i) \neq \bot \\
= & \bigsqcup \{ \mathsf{on}_i(\pi\, w) \mid \mathsf{on}_i(w) = v_i \} & \text{def. } \uparrow \\
\sqsupseteq & (\mathsf{on}_i \circ \pi)(v_1, \dots, v_n) &
\end{array}
$$

Ad 2) $(\forall i)\, \pi_i \neq \perp\!\!\!\perp$ implies $(\forall i)(\exists v_i)\, \pi_i\, v_i \neq \bot$.

$$
\begin{array}{cll}
 & ((\pi_1 \otimes \cdots \otimes \pi_n) \uparrow i)(v) & \\
= & \bigsqcup \{ \mathsf{on}_i((\pi_1 \otimes \cdots \otimes \pi_n)(w)) \mid \mathsf{on}_i(w) = v \} & \text{def. } \uparrow \\
= & \bigsqcup \{ \mathsf{on}_i(\langle \pi_1(\mathsf{on}_1\, w), \dots, \pi_n(\mathsf{on}_n\, w) \rangle) \mid \mathsf{on}_i(w) = v \} & \text{def. } \otimes \\
= & \pi_i(v) & (\forall i)(\exists v_i)\, \pi_i\, v_i \neq \bot \ \blacksquare
\end{array}
$$

The inequality is proper: $\pi_{\text{sor}} \sqsubset (\pi_{\text{sor}} \uparrow 1 \otimes \pi_{\text{sor}} \uparrow 2) = \top\!\!\!\top_{\oplus} \otimes \top\!\!\!\top_{\bot}$ (see below).

To get an idea of the complexity of $\|D_1 \otimes \cdots \otimes D_n\|$ consider the domain $\mathbf{2}_\bot \otimes \mathbf{2}_\bot$ which is the smallest, non-trivial smash product. Using Lemma 4.8 we may characterize its lattice of projections as follows: $\|\mathbf{2}_\bot \otimes \mathbf{2}_\bot\| \cong \|(\mathbf{2} \times \mathbf{2})_\bot\| \cong \|\mathbf{2} \times \mathbf{2}\| \times \mathbf{2}$. Knowing the shape of $\|\mathbf{2} \times \mathbf{2}\|$ we obtain a diagram for $\|\mathbf{2}_\bot \otimes \mathbf{2}_\bot\|$ by placing two copies of $\|\mathbf{2} \times \mathbf{2}\|$ above each other and connecting corresponding points.



Let us call a projection independent if it can be expressed as a product of projections. There are fourteen projections in all, ten of which are independent. [Since $\perp\!\!\!\perp_{\oplus} \otimes \pi = \perp\!\!\!\perp = \pi \otimes \perp\!\!\!\perp_{\oplus}$ for all π, the number of independent projections amounts to a total of $(4-1)*(4-1)+1 = 10$.] The next lemma formally introduces independent projections by means of a closure operation.

4.14 Lemma *The map* $\mathbf{i}(\pi) = \pi \uparrow 1 \otimes \cdots \otimes \pi \uparrow n$ *is a closure operator on* $\|D_1 \otimes \cdots \otimes D_n\|$ *with* $n \geqslant 1$. *A projection is called* independent *iff* $\mathbf{i}(\pi) = \pi$. *The set of all independent projections is denoted by* $\mathbf{i}\,\|D_1 \otimes \cdots \otimes D_n\|$. *Furthermore, we have* $\mathbf{i}\,\|D_1 \otimes \cdots \otimes D_n\| \cong \|D_1\| \otimes \cdots \otimes \|D_n\|$.

Proof Lemma 4.13 shows that $\pi \sqsubseteq \mathbf{i}(\pi)$. Continuity of $\mathbf{i}$ implies monotonicity. It remains to show that $\mathbf{i}$ is idempotent. First note that $\pi \uparrow i = \perp\!\!\!\perp$ if and only if $\pi = \perp\!\!\!\perp$. Let $\pi \neq \perp\!\!\!\perp$, then

$$
\begin{array}{rcll}
\mathbf{i}(\mathbf{i}(\pi)) & = & \mathbf{i}(\pi) \uparrow 1 \otimes \cdots \otimes \mathbf{i}(\pi) \uparrow n & \text{def. } \mathbf{i} \\
 & = & (\pi \uparrow 1 \otimes \cdots \otimes \pi \uparrow n) \uparrow 1 \otimes \cdots \otimes (\pi \uparrow 1 \otimes \cdots \otimes \pi \uparrow n) \uparrow n & \text{def. } \mathbf{i} \\
 & = & \pi \uparrow 1 \otimes \cdots \otimes \pi \uparrow n & (\forall i)\, \pi \uparrow i \neq \perp\!\!\!\perp \\
 & = & \mathbf{i}(\pi). & \text{def. } \mathbf{i}.
\end{array}
$$

The monotonic maps $\pi \mapsto \langle \pi \uparrow 1, \dots, \pi \uparrow n \rangle$ and $v \mapsto \mathsf{on}_1\, v \otimes \cdots \otimes \mathsf{on}_n\, v$ establish an isomorphism between $\mathbf{i}\,\|D_1 \otimes \cdots \otimes D_n\|$ and $\|D_1\| \otimes \cdots \otimes \|D_n\|$. $\blacksquare$

Note that the restriction to non-empty products is necessary since $\mathbf{i}(\perp\!\!\!\perp) = \top\!\!\!\top$ for $n = 0$, that is, the bottom projection is erroneously classified as not independent.

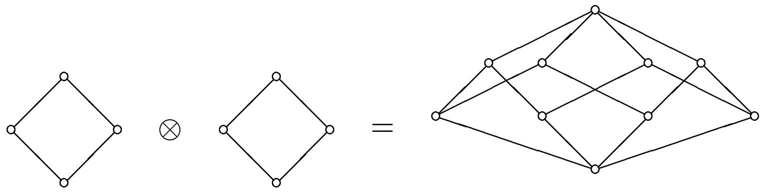
To distinguish between joins and meets on $\|D_1 \otimes \cdots \otimes D_n\|$ and $\mathbf{i}\,\|D_1 \otimes \cdots \otimes D_n\|$ we use the symbols $\dot{\sqcup}$ and $\dot{\sqcap}$ for the operations on the lattice of independent projections. They may be characterized as follows (this is an immediate consequence of Lemma 4.14 using Theorem A.6).

$$
\begin{aligned}
\pi \,\dot{\sqcap}\, \pi' &= \pi \sqcap \pi' \\
\pi \,\dot{\sqcup}\, \pi' &= \mathbf{i}(\pi \sqcup \pi')
\end{aligned}
$$

Note that $\mathbf{i}\,\|D_1 \otimes \cdots \otimes D_n\|$ is *not* a sublattice of $\|D_1 \otimes \cdots \otimes D_n\|$ since, for example, the least upper bound of $\perp\!\!\!\perp_\perp \otimes \top\!\!\!\top_{\obar}$ and $\top\!\!\!\top_{\obar} \otimes \perp\!\!\!\perp_\perp$ is different in each case. In other words, independent projections are not closed under joins.

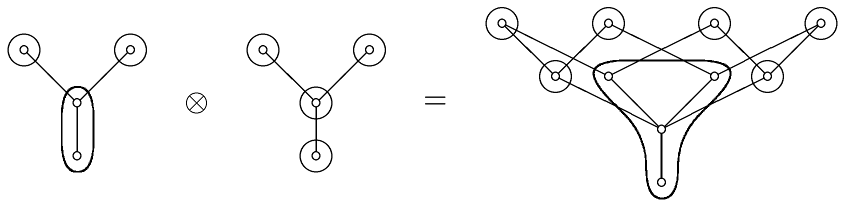
$\pi_1 := \perp\!\!\!\perp_\perp \otimes \top\!\!\!\top_{\obar}$ $\pi_2 := \top\!\!\!\top_{\obar} \otimes \perp\!\!\!\perp_\perp$ $\pi_1 \sqcup \pi_2$ $\pi_1 \,\dot{\sqcup}\, \pi_2$

The shaded elements in the Hasse diagram of $\mathbf{2}_\perp \otimes \mathbf{2}_\perp$ above identify the independent projections. Using Lemma 4.14 the lattice of independent projections may be constructed more systematically.



$\|\mathbf{2}_\perp\|$ $\|\mathbf{2}_\perp\|$ $\mathbf{i}\,\|\mathbf{2}_\perp \otimes \mathbf{2}_\perp\|$

The use of independent projections is vital for backward strictness analysis because they allow to propagate projections backward down expressions. Unfortunately, one looses information as Lemma 4.13 shows. Factoring the projection π_{sor} into independent projections yields the projection $\top\!\!\!\top_{\obar} \otimes \top\!\!\!\top_\perp$.



$\pi_{\mathrm{sor}} \uparrow 1$ $\pi_{\mathrm{sor}} \uparrow 2$ $\pi_{\mathrm{sor}} \uparrow 1 \otimes \pi_{\mathrm{sor}} \uparrow 2$

The information present in π_{sor}, that the result of 'sequential or' is undefined if the first argument equals **f** and the second $\perp$, and that the value of the second argument does not matter, if the first evaluates to **t**, is lost by the factorization.

In order to represent the whole of $\|D_1 \otimes \cdots \otimes D_n\|$ we may resort to Theorem 4.6. Fortunately, the atoms of this projection lattice are independent.

4.15 Fact *Let $D_1, \ldots, D_n$ be domains and let $x_i \in F^\circ(D_i)$, for $1 \leqslant i \leqslant n$, then*

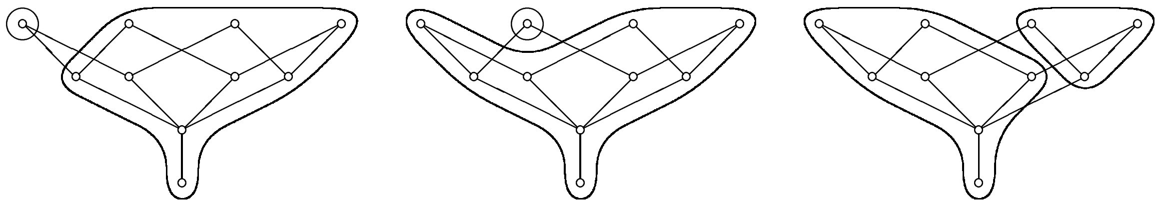
$$[(x_1, \ldots, x_n)] = [x_1] \otimes \cdots \otimes [x_n].$$

Consequently every projection on a smash product may be realized by a set of independent projections.

4.16 Corollary *Let* $\pi \in \|D_1 \otimes \cdots \otimes D_n\|$, *then*

$$\pi \;=\; \bigsqcup\{\, \pi_1 \otimes \cdots \otimes \pi_n \mid \pi_1 \otimes \cdots \otimes \pi_n \sqsubseteq \pi \,\}.$$

Returning to our running example we see that the projection π_{sor} may be represented by the set $\{[\textbf{up f}] \otimes [\textbf{up f}], [\textbf{up f}] \otimes [\textbf{up t}], [\textbf{up t}] \otimes [\textbf{up}\ \bot]\}$ which contains the three non-trivial fixpoints of π_{sor}.



$[\textbf{up f}] \otimes [\textbf{up f}]$ $\qquad$ $[\textbf{up f}] \otimes [\textbf{up t}]$ $\qquad$ $[\textbf{up t}] \otimes [\textbf{up}\ \bot]$

A backward analyser which employs only independent projections amounts to an independent attribute analysis. An analysis which uses the whole of $\|D_1 \otimes \cdots \otimes D_n\|$ is accordingly classified as relational (see Section 1.3).

Note that Corollary 4.16 does *not* apply to normal products. [This is in contrast to a result by Davis [47], see his Proposition 5.1.] The atom $[11]$ on $\mathbf{2}\times\mathbf{2}$ is, for example, not independent ($[11] \sqsubset [1] \times [1] = \top\!\!\top \times \top\!\!\top$). This failure is due to the fact that the set of independent projections on $D_1 \times \cdots \times D_n$ is closed under joins. Products and smash products behave in some sense complementary.

closed wrt	&	$\sqcup$
$\mathbf{i}\,\|D_1 \times \cdots \times D_n\|$	no	yes
$\mathbf{i}\,\|D_1 \otimes \cdots \otimes D_n\|$	yes	no

Note, too, that independent projections on smash products are *not* completely independent since a smash tuple collapses if only a single argument is undefined.

4.5 Projections and strictness

Projections have been introduced for the specification of context dependent strictness properties. The definition below introduces the central notion of 'π_2-strict in context π_1'. The aim of this section is then to relate classical notions of strictness such as 'strict in the first argument' or 'bistrict' to strictness in the sense of this Definition.

4.17 Definition *Let* $\varphi : D \to E$, *let* $\pi_1 \in \|E\|$, *and let* $\pi_2 \in \|D\|$. *Then* φ *is called* π_2*-strict in context* π_1 *iff* $\pi_1 \circ \varphi = \pi_1 \circ \varphi \circ \pi_2$ *(or equivalently* $\pi_1 \circ \varphi \sqsubseteq \varphi \circ \pi_2$*).*

Note that $\bot\!\!\bot \circ \varphi \sqsubseteq \varphi \circ \pi$ and $\pi \circ \varphi \sqsubseteq \varphi \circ \top\!\!\top$ hold trivially, for each π. Furthermore, let $\pi_1' \sqsubseteq \pi_1$ and $\pi_2 \sqsubseteq \pi_2'$, then $\pi_1 \circ \varphi \sqsubseteq \varphi \circ \pi_2$ implies $\pi_1' \circ \varphi \sqsubseteq \varphi \circ \pi_2'$.

The following definition formulates various properties of functions.

4.18 Definition *Let* $\varphi : D_\bot \circ\!\!\to E$ *be a strict function,*

1. *φ is called lift strict iff $\varphi(\mathbf{up}\,\bot) = \bot$,*

2. *φ is called divergent iff $(\forall v \in D)\, \varphi(\mathbf{up}\, v) = \bot$,*

3. *φ ignores its argument iff $(\forall v \in D)\, \varphi(\mathbf{up}\, v) = \varphi(\mathbf{up}\,\bot)$.*

Note that we speak of lift strictness rather than strictness in the sequel, since this is in accordance with our preference of the function space $D_\bot \multimap E$ to $D \to E$.

Ironically, a function is divergent iff it is lift strict and ignores its argument. In other words, the implication $v = \bot \implies \varphi(\mathbf{up}\, v) = \bot$ is not necessarily causal, that is, the non-termination of $\varphi(\mathbf{up}\, v)$ need not be attributed to the non-termination of the argument v.

Each of the properties defined in 4.18 may be recasted in terms of projections.

4.19 Lemma *Let $\varphi : D_\bot \multimap E$ be a strict function,*

1. *φ is lift strict iff $\varphi = \varphi \circ \top\!\!\top_{\oplus}$,*

2. *φ is divergent iff $\varphi = \varphi \circ \bot\!\!\bot_{\oplus}$,*

3. *φ ignores its argument iff $\varphi = \varphi \circ \bot\!\!\bot_{\bot}$.*

Proof Ad 1) '$\Longrightarrow$': Since φ and $\top\!\!\top_{\oplus}$ are strict, the result holds trivially if the argument is improper. Case $\mathbf{up}\,\bot$:

$$\varphi(\mathbf{up}\,\bot) = \bot = \varphi(\bot) = \varphi(\top\!\!\top_{\oplus}(\mathbf{up}\,\bot)) = (\varphi \circ \top\!\!\top_{\oplus})(\mathbf{up}\,\bot).$$

Case $\mathbf{up}\, v^\circ$ with $v^\circ \in D^\circ = D \setminus \{\bot\}$:

$$\varphi(\mathbf{up}\, v^\circ) = \varphi(\top\!\!\top_{\oplus}(\mathbf{up}\, v^\circ)) = (\varphi \circ \top\!\!\top_{\oplus})(\mathbf{up}\,\bot).$$

'$\Longleftarrow$': The proposition follows by a simple calculation.

$$\varphi(\mathbf{up}\,\bot) = (\varphi \circ \top\!\!\top_{\oplus})(\mathbf{up}\,\bot) = \varphi(\bot) = \bot$$

Ad 2 and 3) Using $\bot\!\!\bot_{\oplus}(\mathbf{up}\, v) = \bot$ and $\bot\!\!\bot_{\bot}(\mathbf{up}\, v) = \mathbf{up}\,\bot$, respectively. ■

Comparing the two characterizations of lift strictness we see that the formalization has been raised from an object level where we speak of individual elements to a functional level where the property is expressed in terms of functional composition. [A similar separation is to be found in domain theory where on distinguishes between internal and external axiomatization.]

Things become more interesting if we consider binary functions.

4.20 Definition *Let $\varphi : D_\bot \otimes E_\bot \multimap F$ be a strict function,*

1. *φ is called lift strict in its first argument iff $(\forall w \in E)\, \varphi(\mathbf{up}\,\bot, \mathbf{up}\, w) = \bot$,*

2. *φ is called bistrict iff $(\forall v \in D, w \in E)\, v = \bot \lor w = \bot \implies \varphi(\mathbf{up}\, v, \mathbf{up}\, w) = \bot$,*

3. *φ is called joint strict iff $(\forall v \in D, w \in E)\, v = \bot \land w = \bot \implies \varphi(\mathbf{up}\, v, \mathbf{up}\, w) = \bot$,*

4. *φ ignores its first argument iff $(\forall v \in D, w \in E)\, \varphi(\mathbf{up}\, v, \mathbf{up}\, w) = \varphi(\mathbf{up}\,\bot, \mathbf{up}\, w)$.*

Lift strictness and absence in the second argument are defined accordingly.

The notions introduced are not independent. A function $\varphi : D_\perp \otimes E_\perp \circ\!\!\rightarrow F$ is bistrict iff it is lift strict in its first and its second argument. Let $\hat{\varphi} : (D \times E)_\perp \circ\!\!\rightarrow F$ be the isomorphic image of φ (recall that $D_\perp \otimes E_\perp \circ\!\!\rightarrow F \cong (D \times E)_\perp \circ\!\!\rightarrow F$). Then φ is joint strict iff $\hat{\varphi}$ is lift strict.

Again we may express these properties in terms of projections.

4.21 Lemma *Let $\varphi : D_\perp \otimes E_\perp \circ\!\!\rightarrow F$ be a strict function,*

1. *φ is lift strict in its first argument iff $\varphi = \varphi \circ (\top\!\!\top_{\textcircled{\tiny\ensuremath{\perp}}} \otimes \top\!\!\top_\perp)$,*
2. *φ is bistrict iff $\varphi = \varphi \circ (\top\!\!\top_{\textcircled{\tiny\ensuremath{\perp}}} \otimes \top\!\!\top_{\textcircled{\tiny\ensuremath{\perp}}})$,*
3. *φ is joint strict iff $\varphi = \varphi \circ (\top\!\!\top_{\textcircled{\tiny\ensuremath{\perp}}} \otimes \top\!\!\top_\perp \sqcup \top\!\!\top_\perp \otimes \top\!\!\top_{\textcircled{\tiny\ensuremath{\perp}}})$,*
4. *φ ignores its first argument iff $\varphi = \varphi \circ (\bot\!\!\bot_\perp \otimes \top\!\!\top_\perp)$.*

Proof Analogous to Lemma 4.19. ∎

Note that using projections we may easily combine different properties, for example, φ satisfying $\varphi = \varphi \circ \bot\!\!\bot_\perp \otimes \top\!\!\top_{\textcircled{\tiny\ensuremath{\perp}}}$ ignores its first and is lift strict in its second argument. Furthermore note that Definition 4.20 and Lemma 4.21 extend to multiary functions as well.

In the definition of the standard semantics we decided to model pairs by smash products $D_\perp \otimes E_\perp$ rather than by lifted products $(D \times E)_\perp$. Comparing the expressive power of independent projections on the two isomorphic domains shows that this was in fact the right choice. Consider the domains $\mathbf{2}_\perp \otimes \mathbf{2}_\perp \cong (\mathbf{2} \times \mathbf{2})_\perp$. Since the strictness analyser to be introduced in Chapter 7 treats predefined domains like $\mathbb{Z}_\perp$ essentially as the two-point order $\mathbf{2}$, the domain $\mathbf{2}_\perp \otimes \mathbf{2}_\perp$ actually models pairs of predefined values. Its projection lattice is displayed in Figure 4.2. First note, that there are $(2 * 2 - 1)^2 + 1 = 10$ independent projections on $\mathbf{2}_\perp \otimes \mathbf{2}_\perp$, but only $(2 * 2) * 2 = 8$ on $(\mathbf{2} \times \mathbf{2})_\perp$. Furthermore the two sets have non-empty differences. The projection $\top\!\!\top_{\textcircled{\tiny\ensuremath{\perp}}} \otimes \top\!\!\top_\perp$, for example, which specifies that the first component is definitely required and that the second component may or may not be required, is not expressible as an independent projection on $(\mathbf{2} \times \mathbf{2})_\perp$. Conversely, the projection $(\top\!\!\top \times \top\!\!\top)_{\textcircled{\tiny\ensuremath{\perp}}}$ is not an element of $\mathbf{i}\,\|\mathbf{2}_\perp \otimes \mathbf{2}_\perp\|$.

We have seen that a function φ satisfying $\varphi = \varphi \circ (\top\!\!\top_{\textcircled{\tiny\ensuremath{\perp}}} \otimes \top\!\!\top_\perp)$ is strict in its first argument, while a function obeying $\varphi = \varphi \circ (\top\!\!\top \times \top\!\!\top)_{\textcircled{\tiny\ensuremath{\perp}}}$ is joint strict. Hence, independent projections on smash products capture the classical notion of strictness while independent projections on products do not. Hence we may conclude that smash products are preferable to ordinary products. [Caveat: The projection $(\top\!\!\top \times \bot\!\!\bot)_{\textcircled{\tiny\ensuremath{\perp}}}$ specifies that the first component is required but the second is *ignored*.]

4.6 Disjunction and conjunction

We have discussed in the introduction to this dissertation that backward strictness analysis may be logically divided into two phases. The operators defined in Sections 4.2 to 4.4 are employed in the first phase, where a demand is propagated from the root of an expression to its leaves. In the second phase the results of different branches are combined and propagated back to the root. This section deals with the operators necessary for the second phase.

If we combine abstract values, we must essentially consider the flow of control. Assume that an expression e contains two subexpressions. If both subexpressions are definitely evaluated, both must be defined in order to guarantee that e is defined as well. This motivates the following definition.

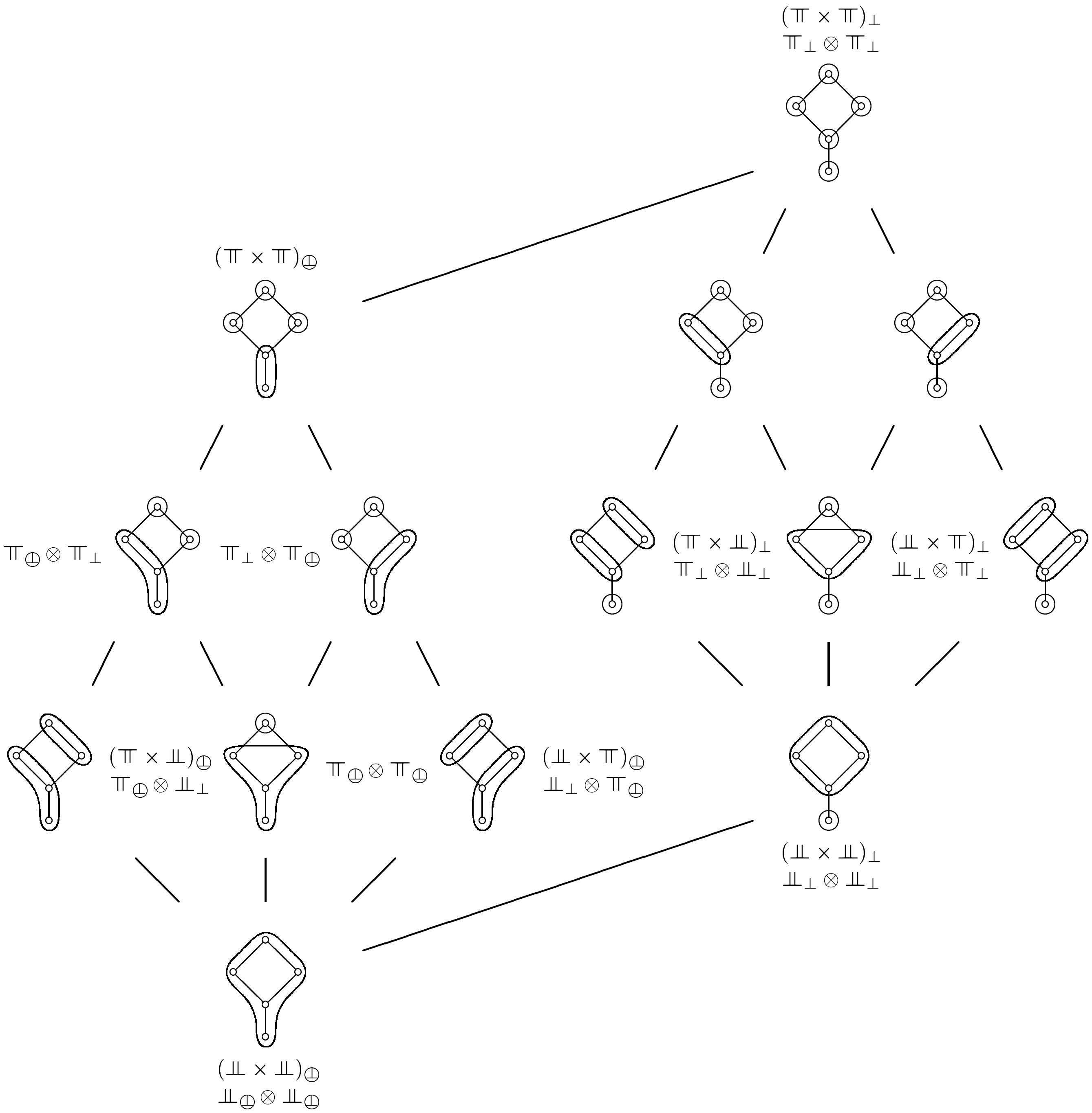


Figure 4.2: Projections on $\mathbf{2}_{\perp} \otimes \mathbf{2}_{\perp}$

4.22 Definition *Let $S \subseteq D$. The* conjunction *of S is defined as follows,*

$$\begin{array}{rcll} \& S & = & \bot & \textbf{if } \bot \in S \\ & = & \bigsqcup S & \textbf{otherwise}. \end{array}$$

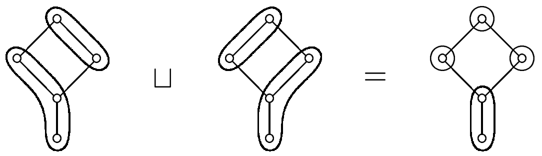
A binary variant of $\&$ is defined by $v_1 \& v_2 = \&\{v_1, v_2\}$. Let $\Pi \subseteq \|D\|$ be a set of projections. Their conjunction *is given by $(\& \Pi)(v) = \& \Pi(v)$. Finally, set $\pi_1 \& \pi_2 = \&\{\pi_1, \pi_2\}$.*

Note that $\&$ is just the strict variant of $\sqcup$. A value is acceptable to $\pi_1 \& \pi_2$ if and only if it is acceptable to π_1 *and* π_2 (motivating the term ‘conjunction of demands’).

Conversely, evaluating exactly one of the subexpressions implies that either of them must be defined. The disjunction of demands is, of course, modeled by the join of projections. A value is acceptable to $\pi_1 \sqcup \pi_2$ if and only if it is acceptable to π_1 *or* π_2. The following diagram may serve as an *aide memoire*.

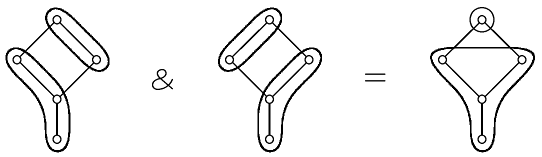
$\pi_1 \& \pi_2$ $\qquad\qquad$ $\pi_1 \sqcup \pi_2$

The operators $\&$ and $\sqcup$ have appealing graphical interpretations. Each block of $\pi_1 \sqcup \pi_2$ is obtained as the intersection of two overlapping blocks in π_1 and π_2, respectively.

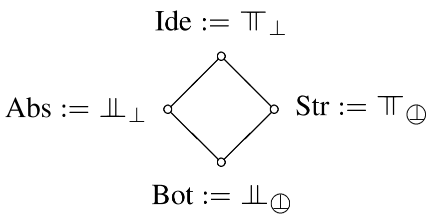


Since each block of $\top\!\!\top_{\oplus} \otimes \perp\!\!\perp_{\bot}$ overlaps with each block in $\perp\!\!\perp_{\bot} \otimes \top\!\!\top_{\oplus}$, their join consists of $2 * 2 = 4$ blocks. [More generally, the blocks of π_2 with $\pi_1 \sqsubseteq \pi_2$ are contained in the blocks of π_1. Assume that $\pi_2(v) = \pi_2(w)$. Since blocks are convex $\pi_1(v) \sqsubseteq \pi_2(v) \sqsubseteq v$ and $\pi_1(w) \sqsubseteq \pi_2(w) \sqsubseteq w$ imply $\pi_1(v) = \pi_1(\pi_2(v)) = \pi_1(\pi_2(w)) = \pi_1(w)$.]

The graphical construction of $\pi_1 \& \pi_2$ is equally simple: The blocks containing the bottom element $\bot$ are merged, the remaining blocks are intersected.



The example shows that the conjunction $\pi_1 \& \pi_2$ is in general non-comparable to both π_1 and π_2. On $\|\mathbf{2}_\bot\|$ the conjunction yields the following results.



$\&$	Bot	Str	Abs	Ide
Bot	Bot	Bot	Bot	Bot
Str	Bot	Str	Str	Str
Abs	Bot	Str	Abs	Ide
Ide	Bot	Str	Ide	Ide

Conjunction and disjunction satisfy a large number of algebraic identities, see below. Half of the equations state lattice-theoretic properties of the join operation. They are listed primarily for the sake of completeness.

4.23 Lemma *Disjunction $\sqcup$ and conjunction $\&$ of demands satisfy,*

1. $(\pi_1 \sqcup \pi_2) \sqcup \pi_3 \;=\; \pi_1 \sqcup (\pi_2 \sqcup \pi_3)$ *(associative laws)*
   $(\pi_1 \,\&\, \pi_2) \,\&\, \pi_3 \;=\; \pi_1 \,\&\, (\pi_2 \,\&\, \pi_3)$
2. $\pi_1 \sqcup \pi_2 \;=\; \pi_2 \sqcup \pi_1$ *(commutative laws)*
   $\pi_1 \,\&\, \pi_2 \;=\; \pi_2 \,\&\, \pi_1$
3. $\pi \sqcup \pi \;=\; \pi$ *(idempotency laws)*
   $\pi \,\&\, \pi \;=\; \pi$
4. $\top\!\!\!\top \sqcup \pi \;=\; \top\!\!\!\top$ *(zero elements)*
   $\perp\!\!\!\perp \,\&\, \pi \;=\; \perp\!\!\!\perp$
5. $\perp\!\!\!\perp \sqcup \pi \;=\; \pi$ *(unit elements)*
   $\perp\!\!\!\perp_{\perp} \,\&\, \pi \;=\; \pi$
6. $\pi_1 \,\&\, (\pi_2 \sqcup \pi_3) \;=\; \pi_1 \,\&\, \pi_2 \sqcup \pi_1 \,\&\, \pi_3$ *(distributive laws)*
   $(\pi_1 \sqcup \pi_2) \,\&\, \pi_3 \;=\; \pi_1 \,\&\, \pi_3 \sqcup \pi_2 \,\&\, \pi_3.$
7. $\pi_1 \sqcup \pi_2 \;=\; \pi_1 \sqcup \pi_1 \,\&\, \pi_2 \sqcup \pi_2.$ *(inclusion law)*

Proof It suffices to show the equations for $\&$, since the equations for $\sqcup$ state its lattice-theoretic properties. Properties 1, 2, 3, and 4 follow immediately from the definition. Ad 5) Case $\pi(\mathbf{up}\, v) = \perp$: Trivial. Case $\pi(\mathbf{up}\, v) \neq \perp$:

$$(\perp\!\!\!\perp_{\perp} \,\&\, \pi)(\mathbf{up}\, v) = \perp\!\!\!\perp_{\perp}(\mathbf{up}\, v) \sqcup \pi(\mathbf{up}\, v) = \mathbf{up}\perp \sqcup\, \pi(\mathbf{up}\, v) = \pi(\mathbf{up}\, v).$$

Ad 6) If $\pi_1(v) = \perp$ or $\pi_2(v) = \perp = \pi_3(v)$, then both sides evaluate to $\perp$. Otherwise they equal $\pi_1(v) \sqcup \pi_2(v) \sqcup \pi_3(v)$. Ad 7) The definition of $\&$ implies $\pi_1 \,\&\, \pi_2 \sqsubseteq \pi_1 \sqcup \pi_2$. Using $\pi_1 \sqsubseteq \pi_2 \iff \pi_1 \sqcup \pi_2 = \pi_2$ the proposition follows. ∎

We will show in Section 6.1 (cf Theorem 6.9) that this set of equations is even complete in the sense that each equation between projections, composed of projection variables, $\perp\!\!\!\perp$ and the operators $\&$ and $\sqcup$, which is valid in every projection lattice turns out to be a logical consequence of the axioms above.

Note that the equation $\perp\!\!\!\perp_{\perp} \,\&\, \pi = \pi$ is not completely general as π is restricted to projections on a lifted domain. We will improve on this in Section 5.3 where we characterize conjunctive units for our standard set of domain operators.

The distributive laws generalize to the infinite case.

4.24 Join-Infinite Distributive Law *Let $\Pi \subseteq \|D\|$ be a set of projections, then*

$$\pi \,\&\, \bigsqcup \Pi \;=\; \bigsqcup \pi \,\&\, \Pi.$$

Proof If $\pi(v) = \perp$ or $(\forall \pi \in \Pi)\ \pi(v) = \perp$, then both sides evaluate to $\perp$. Otherwise they equal $\pi(v) \sqcup \bigsqcup \Pi(v)$. ∎

It remains to relate conjunction and disjunction of demands with the operators introduced in Section 4.2, 4.3, and 4.4 respectively. Lemma 4.2 shows that every projection on a lifted domain $D_\perp$ is of the form π_ℓ with $\ell \in \{\phi, \perp\}$ and $\pi \in \|D\|$. Consequently, we may convert conjunctions and disjunctions of projections on $D_\perp$ into this simple representation. Since $\|D_\perp\|$ is isomorphic to $\|D\| \times \mathbf{2}$ the join is formed coordinatewise: $\pi_\ell \sqcup \pi'_{\ell'} = (\pi \sqcup \pi')_{\ell \sqcup \ell'}$. If the lift of the operands are equal, their conjunction yields: $\pi_\phi \,\&\, \pi'_\phi = (\pi \,\&\, \pi')_\phi$ and $\pi_\perp \,\&\, \pi'_\perp = (\pi \sqcup \pi')_\perp$. The latter equation holds because $\pi_\perp$ maps only $\perp$ onto $\perp$. If the lifts are different the conjunction propagates half-heartily: $\pi_\phi \,\&\, \pi'_\perp = (\pi \sqcup \pi \,\&\, \pi')_\phi$. The disjunctive term $\pi \sqcup \cdot$ may be motivated as follows: Since we merely know that π's worth is definitely needed, the unlifted result must not be smaller than π.

On $\|D_1 \oplus \cdots \oplus D_n\|$ and **i** $\|D_1 \otimes \cdots \otimes D_n\|$ conjunction and disjunction can be characterized in a straightforward manner.

4.25 Lemma *Conjunction and disjunction interact with $(\cdot)_\phi$, $(\cdot)_\perp$, $\oplus$, $\downarrow$, $\otimes$ and $\uparrow$ as follows,*

1. $$\begin{aligned} \pi_\phi \sqcup \pi'_\phi &= (\pi \sqcup \pi')_\phi & \pi_\phi \,\&\, \pi'_\phi &= (\pi \,\&\, \pi')_\phi \\ \pi_\phi \sqcup \pi'_\perp &= (\pi \sqcup \pi')_\perp & \pi_\phi \,\&\, \pi'_\perp &= (\pi \sqcup \pi \,\&\, \pi')_\phi \\ \pi_\perp \sqcup \pi'_\phi &= (\pi \sqcup \pi')_\perp & \pi_\perp \,\&\, \pi'_\phi &= (\pi \,\&\, \pi' \sqcup \pi')_\phi \\ \pi_\perp \sqcup \pi'_\perp &= (\pi \sqcup \pi')_\perp & \pi_\perp \,\&\, \pi'_\perp &= (\pi \sqcup \pi')_\perp \end{aligned}$$

2. $$\begin{aligned} (\pi_1 \oplus \cdots \oplus \pi_n) \sqcup (\pi'_1 \oplus \cdots \oplus \pi'_n) &= (\pi_1 \sqcup \pi'_1) \oplus \cdots \oplus (\pi_n \sqcup \pi'_n) \\ (\pi_1 \oplus \cdots \oplus \pi_n) \,\&\, (\pi'_1 \oplus \cdots \oplus \pi'_n) &= (\pi_1 \,\&\, \pi'_1) \oplus \cdots \oplus (\pi_n \,\&\, \pi'_n) \end{aligned}$$

3. $$\begin{aligned} \pi \downarrow i \sqcup \pi' \downarrow i &= (\pi \sqcup \pi') \downarrow i \\ \pi \downarrow i \,\&\, \pi' \downarrow i &= (\pi \,\&\, \pi') \downarrow i \end{aligned}$$

4. *Let $\pi_i^\circ \neq \perp\!\!\!\perp$ be a projection on D_i, then*
$$\begin{aligned} (\pi_1^\circ \otimes \cdots \otimes \pi_n^\circ) \,\dot{\sqcup}\, (\pi_1^{\circ\prime} \otimes \cdots \otimes \pi_n^{\circ\prime}) &= (\pi_1^\circ \sqcup \pi_1^{\circ\prime}) \otimes \cdots \otimes (\pi_n^\circ \sqcup \pi_n^{\circ\prime}) \\ (\pi_1 \otimes \cdots \otimes \pi_n) \,\&\, (\pi'_1 \otimes \cdots \otimes \pi'_n) &= (\pi_1 \,\&\, \pi'_1) \otimes \cdots \otimes (\pi_n \,\&\, \pi'_n) \end{aligned}$$

5. $$\pi \uparrow i \sqcup \pi' \uparrow i \;=\; (\pi \sqcup \pi') \uparrow i \;=\; (\pi \,\dot{\sqcup}\, \pi') \uparrow i.$$

Proof The equations for $\sqcup$ and $\dot{\sqcup}$ with the exception of 5 may be derived using the isomorphisms $\|D_\perp\| \cong \|D\| \times \mathbf{2}$, $\|D_1 \oplus \cdots \oplus D_n\| \cong \|D_1\| \times \cdots \times \|D_n\|$ and **i** $\|D_1 \otimes \cdots \otimes D_n\| \cong \|D_1\| \otimes \cdots \otimes \|D_n\|$. The laws for lift, $\oplus$ and $\otimes$ capture the fact that the join is formed coordinatewise on (smash) product lattices. Thus we confine ourselves to show the equations involving $\&$. Ad 1) Using $\pi_\phi(\mathbf{up}\; v) = \perp \iff \pi(v) = \perp$ we have $\pi_\phi \,\&\, \pi'_\phi = (\pi \,\&\, \pi')_\phi$.

$$\begin{aligned} \pi_\phi \,\&\, \pi'_\perp &= \pi_\phi \,\&\, (\perp\!\!\!\perp_\perp \sqcup \pi'_\phi) && \text{since } \perp\!\!\!\perp_\perp \sqcup \pi'_\phi = \pi'_\perp \\ &= \pi_\phi \,\&\, \perp\!\!\!\perp_\perp \sqcup \pi_\phi \,\&\, \pi'_\phi && \text{distributive law} \\ &= \pi_\phi \sqcup \pi_\phi \,\&\, \pi'_\phi && \perp\!\!\!\perp_\perp \text{ is unit} \\ &= (\pi \sqcup \pi \,\&\, \pi')_\phi \end{aligned}$$

Finally, $v \neq \perp \Longrightarrow \pi_\perp(v) \neq \perp$ implies $\pi_\perp \& \pi'_\perp = (\pi \sqcup \pi')_\perp$. Ad 3) Follows by $\mathbf{out}_i(\mathbf{in}_i\; v) = \perp \iff v = \perp$. Ad 2) Let $\pi = \pi_1 \oplus \cdots \oplus \pi_n$ and $\pi' = \pi'_1 \oplus \cdots \oplus \pi'_n$.

$$\begin{aligned} \pi \,\&\, \pi' &= (\pi \,\&\, \pi') \downarrow 1 \oplus \cdots \oplus (\pi \,\&\, \pi') \downarrow n && \text{Lemma 4.13} \\ &= (\pi \downarrow 1 \,\&\, \pi' \downarrow 1) \oplus \cdots \oplus (\pi \downarrow n \,\&\, \pi' \downarrow n) && \text{by Part 3} \\ &= (\pi_1 \,\&\, \pi'_1) \oplus \cdots \oplus (\pi_n \,\&\, \pi'_n) && \text{Lemma 4.13} \end{aligned}$$

Ad 4) The law follows by $\langle \pi_1\, v_1, \ldots, \pi_1\, v_n\rangle = \bot \iff (\exists i)\, \pi_i\, v_i = \bot$. Ad 5)

$$\begin{array}{cll} & ((\pi \sqcup \pi') \uparrow i)(v) & \\ = & \bigsqcup\{\, \mathsf{on}_i((\pi \sqcup \pi')(w)) \mid \mathsf{on}_i(w) = v\,\} & \text{def. } \uparrow \\ = & \bigsqcup\{\, \mathsf{on}_i(\pi\, w) \sqcup \mathsf{on}_i(\pi'\, w) \mid \mathsf{on}_i(w) = v\,\} & \mathsf{on}_i \text{ is additive} \\ = & \bigsqcup\{\, \mathsf{on}_i(\pi\, w) \mid \mathsf{on}_i(w) = v\,\} \sqcup \bigsqcup\{\, \mathsf{on}_i(\pi'\, w) \mid \mathsf{on}_i(w) = v\,\} & \\ = & (\pi \uparrow i)(v) \sqcup (\pi' \uparrow i)(v) & \text{def. } \uparrow \end{array}$$

Finally, $(\pi \uplus \pi') \uparrow i = \mathbf{i}(\pi \sqcup \pi') \uparrow i = (\pi \sqcup \pi') \uparrow i$. ∎

We have already seen in Section 4.4 that projections on smash products are not closed under joins. Generally, we only have $(\pi_1 \otimes \cdots \otimes \pi_n) \sqcup (\pi'_1 \otimes \cdots \otimes \pi'_n) \sqsubseteq (\pi_1 \sqcup \pi'_1) \otimes \cdots \otimes (\pi_n \sqcup \pi'_n)$. Even if we restrict ourselves to the lattice of independent projections some care is necessary as indicated by Part 4 of the Lemma above. Since $⫫ \otimes ⫪ \uplus ⫪ \otimes ⫫ = ⫫ \otimes ⫫ \sqsubseteq ⫪ \otimes ⫪ = (⫫ \sqcup ⫪) \otimes (⫪ \sqcup ⫫)$ equality only holds under the proviso that the component projections are proper. For similar reasons $\uparrow$ does not distribute over $\&$. Consider, for example, $(([\mathbf{f}] \otimes [\mathbf{t}]) \,\&\, ([\mathbf{f}] \otimes [\mathbf{f}])) \uparrow 1 = ([\mathbf{f}] \otimes ⫫) \uparrow 1 = ⫫ \sqsubseteq [\mathbf{f}] = ([\mathbf{f}] \otimes [\mathbf{t}]) \uparrow 1 \,\&\, ([\mathbf{f}] \otimes [\mathbf{f}]) \uparrow 1$. Generally, only the following inequality holds: $(\pi \,\&\, \pi') \uparrow i \sqsubseteq \pi \uparrow i \,\&\, \pi' \uparrow i$.

4.7 Smash projections

In Section 1.2 we have related ideal-based forward analysis and projection-based backward analysis. An appealing equivalence result was obtained by restricting the abstract values of backward analysis to so-called smash projections. This section provides the formal definitions and proves the correspondence between Scott-closed sets or ideals and smash projections.

4.26 Definition *The projection π is called* smash projection *iff either $\pi(v) = \bot$ or $\pi(v) = v$, for all $v \in D$. The set of all finitary smash projections on D is denoted by* $\mathsf{smash}\, \|D\|$.

The conjunction of demands $\&$ is sometimes mistaken for the meet of projections $\sqcap$. In general we have $\pi_1 \sqcap_{\|D\|} \pi_2 \sqsubseteq \pi_1 \sqcap_{D\to D} \pi_2 \sqsubseteq \pi_1 \,\&\, \pi_2$. Both operations coincide if we restrict π_1 and π_2 to smash projections. For smash projections $\pi_i \sqsubseteq \pi_1 \,\&\, \pi_2$ holds, hence $\pi_1 \,\&\, \pi_2 = \pi_1 \sqcap \pi_2$. Furthermore the set of smash projections is the largest subset of $\|D\|$ satisfying this property.

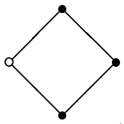
4.27 Lemma *The set of smash projections* $\mathsf{smash}\, \|D\|$ *is the largest set,*

1. *that contains $⫪$ and*
2. *for which $\sqcap$ and $\&$ collapse.*

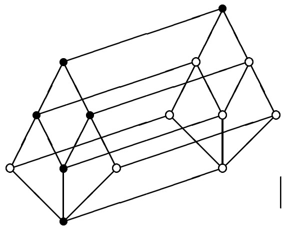
Proof We prove by the way of contradiction that a set satisfying both properties contains only smash projections. Let π be a projection with $\bot \sqsubset \pi(v) \sqsubset v$, for some $v \in D$. It follows that $(\pi \sqcap ⫪)(v) = \pi(v) \sqsubset v = (\pi \,\&\, ⫪)(v)$ ↯. ∎

Bottom and identity are smash projections. Let π_1 and π_2 be smash projections, then $\pi_1 \sqcup \pi_2$ and $\pi_1 \sqcap \pi_2 = \pi_1 \,\&\, \pi_2$ are smash projections as well. Hence $\mathsf{smash}\, \|D\|$ forms a sublattice of $\|D\|$. On $\mathbf{2}_\bot$ there are three, on $\mathbf{2}_\bot \otimes \mathbf{2}_\bot$ six smash projections.

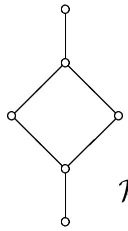
$\mathcal{P}(\mathbf{2}_\perp)$



$\|\mathbf{2}_\perp\|$



$\|\mathbf{2}_\perp \otimes \mathbf{2}_\perp\|$



$\mathcal{P}(\mathbf{2}_\perp \otimes \mathbf{2}_\perp)$

The shaded elements identify the smash projections. Note that they primarily specify classical strictness properties such as 'strict in the first argument'. Since $\perp\!\!\!\perp_\perp(\textbf{up}\,\perp) = \textbf{up}\,\perp$, smash projections fail to express properties such as 'the first argument is ignored'.

The diagrams above exemplify the correspondence between the Hoare or lower powerdomain, $\mathcal{P}(D)$, and the set of all smash projections on D.

4.28 Lemma *Let D be a domain. Then the map $\pi \mapsto \overline{\textbf{im}\,\pi}$ is an order-isomorphism of* **smash** $\|D\|$ *onto $\mathcal{P}(D)$, with its inverse given by $X \mapsto \pi_{\overline{F(X)}}$.*

Proof We show that **smash** $\|D\| \cong \mathcal{F}(F^\circ(D))$. The proposition follows by Theorem A.11 and $\mathcal{O}(P) \cong \mathcal{F}(P)$.

Let $\pi \in$ **smash** $\|D\|$ be a smash projection. We first prove that $M = F^\circ(\textbf{im}\,\pi)$ is an order filter. Let $a \in M$ and $a \sqsubseteq b \in F^\circ(D)$. Since $\perp \sqsubset \pi(a) \sqsubseteq \pi(b)$ and $\pi(b) = \perp \vee \pi(b) = b$ we have $b \in M$.

Conversely, let $M \in \mathcal{F}(F^\circ(D))$ be an order filter. To see that $N = M \cup \{\perp\}$ is normal, let $F \Subset N \cap {\downarrow}v$, for $v \in D$. As $\bigsqcup F \in N \cap {\downarrow}v$, the set $N \cap {\downarrow}v$ is directed. It is easy to see that π_M is a smash projection. Consequently, the monotonic maps $\pi \mapsto F^\circ(\textbf{im}\,\pi)$ and $M \mapsto \pi_M$ establish an isomorphism between $\langle$**smash** $\|D\|; \sqsubseteq\rangle$ and $\langle \mathcal{F}(F^\circ(D)); \subseteq\rangle$. ■

A simple projection which is used in the sequel is π_M with $M = F^\circ(D) \setminus {\downarrow}v$, for some $v \in D$. Since M is a filter, Lemma 4.28 implies that π_M is a smash projection with $F(\textbf{im}\,\pi_M) = M \cup \{\perp\}$.

4.8 Bibliographic notes

Projections respectively projection-embedding pairs were invented by Scott for solving recursive domain equations, see [64] for an excellent overview and for further pointers to the literature. Wadler and Hughes [150] introduced them into the realm of strictness analysis. Many results listed in this chapter are to be found in *loc. cit.*, among others the characterization of strictness, absence, and divergence in terms of projections, the definition of $\&$ and its properties and the relationship to the inverse image analysis of Dybjer [54]. The restriction to finitary projections goes back to Davis [47] (though he states that "no essential use is made of this fact"). Furthermore, the isomorphism $\|D_\perp\| \cong \|D\| \times \mathbf{2}$ is mentioned in *loc. cit.* Part of the notation ($(\cdot)_{\oplus}$ and $(\cdot)_\perp$) and many of the algebraic properties relating $\sqcup$ and $\&$ with $(\cdot)_{\oplus}$, $(\cdot)_\perp$, $\oplus$, and $\otimes$ are due to Kubiak *et. al.* [105].

Chapter 5

Non-standard Semantics

Author's Prayer:
"Copy from one, it's plagiarism; copy from two, it's research."
— Wilson Mizner

Vote for the man who promises least;
he'll be the least disappointing.
— Bernard Baruch, *In Meyer Berger New York (1960)*

Knowing 'everything' about projections we turn now to the definition of the non-standard semantics for the language introduced in Section 2. The non-standard semantics of a function, also called abstraction, is a function which maps demands on the result to demands on the argument. Since the non-standard semantics serves as a connection link between the standard and the approximating semantics we will especially be interested in theoretical questions such as

1. Does a first-order function possess a *least* abstraction?
2. Is the non-standard semantics equivalent to the standard one?

The first point addresses the question of optimality. The smaller an abstraction the larger is its information content. Consequently, a least abstraction is the most informative one with respect to strictness properties. We shall first tackle this question in a semantic setting, that is, we consider continuous functions on domains. In a second step we show how to apply the results to our first-order language.

The non-standard semantics is later approximated for reasons of computability. It is therefore interesting to ask whether this particular model of strictness already involves approximations. A positive answer to the second question may then be regarded as an affirmation that the approach chosen is in fact the right one.

Plan of the chapter Section 5.1 proves that every strict first-order function does possess a least abstraction and that this abstraction in turn determines the function. Unfortunately, least abstractions are not necessarily compositional. Section 5.2 shows that this 'defect' may be repaired by focusing on the class of stable functions. Sections 5.3 to 5.6 examine the different constructions found in the standard semantics and show how least abstractions may be defined for them. Finally, Section 5.7 introduces the non-standard semantics putting together the results of the previous sections.

5.1 Least abstractions

Formally the non-standard semantics of a function is a certain projection transformer.

5.1 Definition *A function* $\tau : \|E\| \to \|D\|$ *is called a* projection transformer. *Let* $\varphi : D \to E$, *then* τ *is an* abstraction of φ *iff* φ *is* $\tau(\pi)$*-strict in context* π, *for all* $\pi \in \|E\|$.

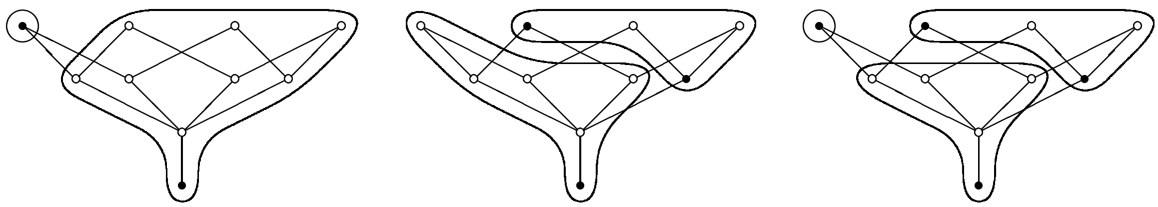
By Definition 4.17 the projection transformer τ is an abstraction of φ iff $\pi \circ \varphi \sqsubseteq \varphi \circ \tau(\pi)$, for all $\pi \in \|E\|$. This condition is sometimes termed 'safety condition' (the terminology is due to the framework of abstract interpretation). A projection transformer which trivially satisfies the safety condition is $\boldsymbol{\lambda}\pi.\top\!\!\top$. However, this abstraction is virtually useless since it carries no information whatsoever. We are, of course, interested in abstractions as small as possible. The functions $\boldsymbol{\lambda}v.\bot$ and $\boldsymbol{\lambda}v.v$, for example, possess the *least* abstractions $\boldsymbol{\lambda}\pi.\bot\!\!\bot$ and $\boldsymbol{\lambda}\pi.\pi$ respectively. The question naturally arises as to whether there is always a least projection transformer satisfying the safety condition. The answer is in the affirmative provided the domain of the function satisfies Axiom I. If the function is strict, the least abstraction in turn determines the function φ. In other words, the least abstraction contains as much information as the original function.

Let us first show why the restriction to strict functions is sensible. The following argument may be seen as a late justification of the call-by-value transformation introduced in Section 2.3. Take as an example the constant function $\boldsymbol{\lambda}v.c$. The least abstraction of $\boldsymbol{\lambda}v.c$ is $\boldsymbol{\lambda}\pi.\bot\!\!\bot$ irrespective of the constant c. Hence in the case of non-strict functions the least abstraction neither determines the function nor is the least abstraction able to detect simple strictness ($\boldsymbol{\lambda}v.c$ is strict if $c = \bot$ and non-strict otherwise). Let $\varphi : D \to E$ be an arbitrary function. Though its strict, lifted counterpart $\varphi^\dagger : D_\bot \circ\!\!\to E$ with $\varphi^\dagger = \textsf{\textbf{strict}}(\varphi \circ \textsf{\textbf{down}})$ does not contain more information, its abstraction which is of type $\|E\| \to \|D_\bot\|$ as opposed to $\|E\| \to \|D\|$ obviously does. The least abstraction of $(\boldsymbol{\lambda}v.c)^\dagger$, for example, is τ_c with

$$\begin{aligned} \tau_c(\pi) &= \bot\!\!\bot_{\oplus} && \textsf{\textbf{if }} \pi(c) = \bot \\ &= \bot\!\!\bot_{\bot} && \textsf{\textbf{otherwise}}. \end{aligned}$$

That is why we transform non-strict programs into equivalent strict ones (cf Section 2.3). Note that we may reconstruct the strictness of φ, since φ is strict if and only if $\varphi^\dagger$ is lift strict.

As usual we start with a graphical interpretation of the problem (of determining the least abstraction for a given function). For this purpose it is easier to use the condition $\pi \circ \varphi = \pi \circ \varphi \circ \tau(\pi)$ rather than the equivalent $\pi \circ \varphi \sqsubseteq \varphi \circ \tau(\pi)$. We further simplify matters by treating the composition $\pi \circ \varphi$ as a unit. Thus it suffices to find the least projection $\hat{\pi}$ satisfying $\hat{\varphi} = \hat{\varphi} \circ \hat{\pi}$, for a given $\hat{\varphi}$. In the case of 'sequential or' we must consequently consider four subproblems (in fact only three since the least projection satisfying $\bot\!\!\bot \circ \text{sor} = \bot\!\!\bot \circ \text{sor} \circ \pi$ is trivially $\bot\!\!\bot$, in other words least abstractions are strict). In the picture below the graphs of the functions $\pi \circ \text{sor}$ with $\pi \in \|\mathbf{2} \oplus \mathbf{2}\|$ are displayed. Recall that a collections of points is encircled iff they have a common image.

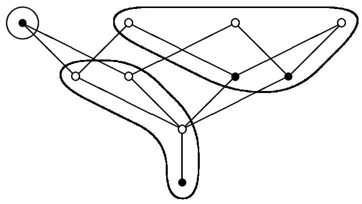


$[\textsf{\textbf{f}}] \circ \text{sor}$ $\quad$ $[\textsf{\textbf{t}}] \circ \text{sor}$ $\quad$ $([\textsf{\textbf{f}}] \sqcup [\textsf{\textbf{t}}]) \circ \text{sor}$

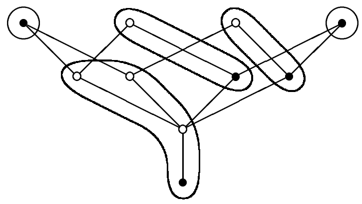
It is quite obvious that any projection must contain the minimal elements (the shaded points) of each block in its image. Let M be set of these elements. Since the remaining points—at least in the examples above—may be mapped to one of the minimal elements, the projection π_M is in fact the least projection satisfying the safety condition. Consequently sor$^\sharp$ with

$$\begin{array}{lcl}
\mathrm{sor}^\sharp(\perp\!\!\!\perp) & = & \perp\!\!\!\perp \\
\mathrm{sor}^\sharp([\mathbf{f}]) & = & [(\mathbf{up}\,\mathbf{f}, \mathbf{up}\,\mathbf{f})] \\
\mathrm{sor}^\sharp([\mathbf{t}]) & = & [(\mathbf{up}\,\mathbf{f}, \mathbf{up}\,\mathbf{t})] \sqcup [(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\perp)] \\
\mathrm{sor}^\sharp([\mathbf{f}] \sqcup [\mathbf{t}]) & = & [(\mathbf{up}\,\mathbf{f}, \mathbf{up}\,\mathbf{f})] \sqcup [(\mathbf{up}\,\mathbf{f}, \mathbf{up}\,\mathbf{t})] \sqcup [(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\perp)]
\end{array}$$

is the least abstraction of ‘sor’. In this case the minimal elements are even the only fixpoints of the corresponding projections. This is not necessarily so. Consider as another example the parallel variant of the logical or.



parallel or



π_{por}

The inverse image of $\mathbf{t}$ has two minimal elements $(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\perp)$ and $(\mathbf{up}\,\perp, \mathbf{up}\,\mathbf{t})$ respectively. Hence the corresponding projection π_{por} contains both points in its image. Lemma A.13 implies that their least upper bound $(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\mathbf{t})$ is a fixpoint as well. [Davis [47] erroneously takes ‘parallel or’ as an example for a function having no least, but two minimal projections satisfying $\mathrm{por} = \mathrm{por} \circ \pi$: a projection π_1 which maps $(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\mathbf{t})$ to $(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\perp)$ and another projection π_2 which maps $(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\mathbf{t})$ to $(\mathbf{up}\,\perp, \mathbf{up}\,\mathbf{t})$. These projections exist but they violate the safety condition since by monotonicity $\pi_1(\mathbf{up}\,\perp, \mathbf{up}\,\mathbf{t}) = (\mathbf{up}\,\perp, \mathbf{up}\,\perp) = \pi_2(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\perp)$.]

Now, let us make the ideas mathematically precise. The following definition will be handy.

5.2 Definition *Let* $\varphi : D \to E$, $x \in D$ *and* $y \in E$, *such that* $y \sqsubseteq \varphi(x)$. *Then*

$$M(\varphi, x, y) \quad := \quad \mathrm{Min}\{\, v \sqsubseteq x \mid y \sqsubseteq \varphi(v) \,\}.$$

An element $m \in M(\varphi, x, y)$ represents the minimal information needed from x to reach y by applying the function φ. We have, for example, $M(\mathrm{sor}, \mathbf{t}, (\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\mathbf{t})) = \{(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\perp)\}$ and $M(\mathrm{por}, \mathbf{t}, (\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\mathbf{t})) = \{(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\perp), (\mathbf{up}\,\perp, \mathbf{up}\,\mathbf{t})\}$. Note that $v \in M(\varphi, x, \pi(\varphi\, x))$ if and only if $v \sqsubseteq x$, $\pi(\varphi\, x) = \pi(\varphi\, v)$ and there is no smaller element satisfying the two conditions. Moreover $M(\varphi, x, y)$ is non-empty, if the domain D is finite. Consequently $\varphi^\sharp$ with

$$\varphi^\sharp(\pi) \quad = \quad \pi_M \text{ where } M = \bigcup\{\, M(\varphi, x, \pi(\varphi\, x)) \mid x \in D \,\}$$

is the least abstraction of $\varphi : D \to E$ for finite D.

Turning to the infinite case it is tempting to set the fixpoint-set M in the definition above to $\bigcup\{\, M(\varphi, x, \pi(\varphi\, x)) \mid x \in F(D) \,\}$. [The motivation is provided by the facts that a continuous function $\varphi : D \to E$ is determined by its effect on the finite elements of D and a finitary projection by the finite elements of its image.] However, we are faced with two problems.

Firstly, the set $M(\varphi, x, \pi(\varphi\, x))$ with $x \in F(D)$ may not contain minimal elements. It is not hard to construct a counterexample on the domain ω^δ which contains only finite elements. Set, for example, $\varphi(n) = \top$ if $n < \omega$ and $\varphi(\omega) = \bot$. Then $\{\, v \sqsubseteq 0 \mid \top \sqsubseteq \varphi(v) \,\}$ is an infinite descending chain.

Secondly, even if minimal elements exist they may not be finite. The crux is that the set $F(D)$ is not closed under going down. Consider, for example, the domain $\omega{+}1$, where $\omega{+}1$ is finite and ω is not. Both problems disappear if we restrict ourselves to domains which satisfy Axiom I. Such domains enjoy the appealing property that $d \in F(D)$ implies ${\downarrow} d \Subset F(D)$.

5.3 Theorem *Let $\varphi : D \to E$. If D is an I-domain, then $\varphi^\sharp$ with*

$$\varphi^\sharp(\pi) \quad = \quad \pi_M \text{ where } M = \bigcup \{\, M(\varphi, x, \pi(\varphi\, x)) \mid x \in F(D) \,\}$$

is the least abstraction of φ.

Proof First note that π_M is well-defined, since $M \subseteq F(D)$ by Axiom I. By continuity it suffices to check $(\pi \circ \varphi)(a) \sqsubseteq (\varphi \circ \pi_M)(a)$ for $a \in F(D)$. Since $M(\varphi, a, \pi(\varphi\, a))$ is non-empty there is a $m \in M$ with $m \sqsubseteq a$ and $\pi(\varphi\, a) \sqsubseteq \varphi(m)$. Therefore $m \sqsubseteq \pi_M(a)$ implying $\pi(\varphi\, a) \sqsubseteq \varphi(\pi_M(a))$.

To see that π_M is the least abstraction, let $a \in M$ and $\hat\pi \in \|D\|$ with $\pi \circ \varphi \sqsubseteq \varphi \circ \hat\pi$. Thus $\hat\pi(a) \in \{v \sqsubseteq a \mid \pi(\varphi\, a) \sqsubseteq \varphi\, v\}$ and by minimality of a we have $\hat\pi(a) = a$. Finally, $M \subseteq F(\mathbf{im}\, \hat\pi)$ implies $F(\mathbf{im}\, \pi_M) = \mathbf{c}(M) \subseteq \mathbf{c}(F(\mathbf{im}\, \hat\pi)) = F(\mathbf{im}\, \hat\pi)$ and by Theorem A.16 $\pi_M \sqsubseteq \hat\pi$. ■

The condition that D must satisfy Axiom I is not very restrictive, since it is met by every domain obtained as the solution of a first-order domain equation (cf Lemma A.25). Hence, quite a substantial class of functions, in particular functions on finite or flat domains, possess least abstractions.

The least backward abstraction of a function enjoys nice algebraic properties.

5.4 Lemma *Let $\varphi : D \circ\!\!\to E$, let $\Pi \subseteq \|E\|$ and assume that φ has the least abstraction $\varphi^\sharp : \|E\| \circ\!\!\to \|D\|$. Then*

1. $\varphi^\sharp(\bigsqcup \Pi) \quad = \quad \bigsqcup \varphi^\sharp(\Pi)$,
2. $\varphi^\sharp(\& \,\Pi) \quad = \quad \& \,\varphi^\sharp(\Pi)$.

Proof Note that we do not assume that D satisfies Axiom I. Hence we cannot resort to the characterization of $\varphi^\sharp$ in Lemma 5.3.

Ad 1) As $\varphi^\sharp$ is the least abstraction, we have $\varphi^\sharp(\pi_1) = \bigsqcap \{\, \pi_2 \mid \pi_1 \circ \varphi \sqsubseteq \varphi \circ \pi_2 \,\}$. Thus $\varphi^\sharp$ is monotonic implying $\varphi^\sharp(\bigsqcup \Pi) \sqsupseteq \bigsqcup \varphi^\sharp(\Pi)$. We show next that φ is $\bigsqcup \varphi^\sharp(\Pi)$-strict in context $\bigsqcup \Pi$.

$$\begin{array}{rcll}
(\bigsqcup \Pi) \circ \varphi & = & \bigsqcup \Pi \circ \varphi & \text{the join is formed pointwise} \\
& \sqsubseteq & \bigsqcup \varphi \circ \varphi^\sharp(\Pi) & \varphi^\sharp \text{ is an abstraction} \\
& \sqsubseteq & \varphi \circ \bigsqcup \varphi^\sharp(\Pi) & \text{monotonicity of } \varphi
\end{array}$$

Since $\varphi^\sharp(\bigsqcup \Pi)$ is the least projection satisfying $(\bigsqcup \Pi) \circ \varphi \sqsubseteq \varphi \circ \pi$, the proposition follows.

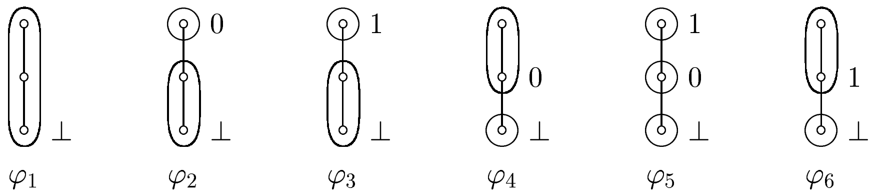
Ad 2) We prove $((\& \Pi) \circ \varphi)(v) \sqsubseteq (\varphi \circ \& \varphi^\sharp(\Pi))(v)$, for all $v \in D$. Case $(\exists \pi \in \Pi)\, \varphi^\sharp(\pi)(v) = \perp$: Strictness of φ implies that $\pi(\varphi(v)) = \perp$. Consequently $(\& \Pi)(\varphi(v)) = \perp$ as well. Case $(\forall \pi \in \Pi)\, \varphi^\sharp(\pi)(v) \neq \perp$: Then

$$((\& \Pi) \circ \varphi)(v) \sqsubseteq ((\bigsqcup \Pi) \circ \varphi)(v) \sqsubseteq (\varphi \circ \bigsqcup \varphi^\sharp(\Pi))(v) = (\varphi \circ \& \varphi^\sharp(\Pi))(v).$$

It remains to show that $\& \varphi^\sharp(\Pi)$ is in fact the least such projection. Case $(\exists \pi \in \Pi)\, \pi(\varphi(v)) = \perp$: Let $M = F(D) \setminus {\downarrow} v$, then $\pi \circ \varphi \sqsubseteq \varphi \circ \pi_M$. Since $\varphi^\sharp$ is the least projection, $\varphi^\sharp(\pi)(v) \sqsubseteq \pi_M(v) = \perp$ and consequently $\& \varphi^\sharp(\Pi)(v) = \perp$. Case $(\forall \pi \in \Pi)\, \pi(\varphi(v)) \neq \perp$: Then $(\& \Pi)(\varphi(v)) = (\bigsqcup \Pi)(\varphi(v))$ and Part 1 implies that $\bigsqcup \varphi^\sharp(\Pi)(v) = \& \varphi^\sharp(\Pi)(v)$ is the least projection. ■

Lemma 5.4 has two immediate consequences. First the least abstraction of a function $\varphi : D \circ\!\!\rightarrow E$ is continuous—provided it exists—and second the least abstraction is determined by the images of $[a]$, for $a \in F^\circ(E)$.

The non-standard semantics of a language is normally quite close to the standard semantics in the sense that the standard semantics may be reconstructed from the non-standard one (take, for instance, the collecting semantics which essentially maps the standard semantics to sets of values). Since backward analysis reverts the usual direction of information flow it is far from clear whether this property holds for backward analysis as well. Let us start with a simple example; consider the strict, monotonic functions on the domain $\mathbf{2}_\perp \cong \mathbf{3}$.



Their abstractions are listed below (by Lemma 5.4 it suffices to list the images of $[0]$ and $[1]$).

π	$\varphi_1^\sharp(\pi)$	$\varphi_2^\sharp(\pi)$	$\varphi_3^\sharp(\pi)$	$\varphi_4^\sharp(\pi)$	$\varphi_5^\sharp(\pi)$	$\varphi_6^\sharp(\pi)$
$[0]$	$\perp\!\!\!\perp$	$[1]$	$[1]$	$[0]$	$[0]$	$[0]$
$[1]$	$\perp\!\!\!\perp$	$\perp\!\!\!\perp$	$[1]$	$\perp\!\!\!\perp$	$[1]$	$[0]$

First thing to note is that the abstractions are pairwise different. Conversely, they are in fact the only *least* abstractions. As $\varphi^\sharp[a]$ must be strictly less than $\top\!\!\!\top$, there are $3^2 = 9$ possible candidates. By Lemma 5.4 we know that $\varphi^\sharp[0] \,\&\, \varphi^\sharp[1] = \varphi^\sharp[1]$, thus ruling out $\{[0] \mapsto [1], [1] \mapsto [0]\}$, $\{[0] \mapsto \perp\!\!\!\perp, [1] \mapsto [0]\}$ and $\{[0] \mapsto \perp\!\!\!\perp, [1] \mapsto [1]\}$.

Now, assume that a least abstraction $\varphi^\sharp$ is given and we want to reconstruct $\varphi : D \circ\!\!\rightarrow E$ from $\varphi^\sharp$. Since E is algebraic $\varphi(v)$ is determined by the least upper bound of its approximations $\bigsqcup\{\, w \in F^\circ(E) \mid w \sqsubseteq \varphi(v) \,\}$. But if $\perp \sqsubset w \sqsubseteq \varphi(v)$ holds, v must be acceptable to $\varphi^\sharp[w]$. Conversely, v being acceptable to $\varphi^\sharp[w]$ implies $w \sqsubseteq \varphi(v)$, because $\varphi^\sharp$ is the least abstraction. Hence it suffices to check the argument, whose image under φ we wish to determine, for acceptability. For example, as 0 is only acceptable to $\varphi_5^\sharp[0]$, $\varphi_5(0)$ must equal 0, on the other hand 1 is acceptable to both $\varphi_5^\sharp[0]$ and $\varphi_5^\sharp[1]$, so we may conclude that $\varphi_5(1) = 0 \sqcup 1 = 1$.

5.5 Theorem *Let $\varphi : D \circ\!\!\rightarrow E$ and assume that φ has a least abstraction $\varphi^\sharp : \|E\| \circ\!\!\rightarrow \|D\|$. For each $v \in D$,*

$$\varphi(v) \quad = \quad \bigsqcup\{\, w \in F^\circ(E) \mid v \text{ is acceptable to } \varphi^\sharp[w] \,\}.$$

Proof Since $\varphi(v) = \bigsqcup\{\, w \in F^\circ(E) \mid w \sqsubseteq \varphi(v) \,\}$ is suffices to show that v is acceptable to $\varphi^\sharp[w]$ iff $w \sqsubseteq \varphi(v)$, for each $w \in F^\circ(D)$. '$\Longleftarrow$': Since φ is strict and and

$$\bot \sqsubset w = ([w] \circ \varphi)(v) \sqsubseteq (\varphi \circ \varphi^\sharp[w])(v),$$

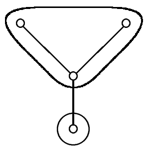
we may conclude that $\varphi^\sharp[w])(v) \neq \bot$. '$\Longrightarrow$': We shall prove the contrapositive. Assume that $w \not\sqsubseteq \varphi(v)$ and let $M = F(D) \setminus \downarrow v$. Since $([w] \circ \varphi)(v) = \bot$ we have $[w] \circ \varphi \sqsubseteq \varphi \circ \pi_M$. This in turn implies $\varphi^\sharp[w](v) \sqsubseteq \pi_M(v) = \bot$, because $\varphi^\sharp$ is the least abstraction of φ. ■

Theorem 5.3 implies that functions formulated in the first-order language of Section 2 always possess least abstractions. [Note that this result holds for an arbitrary set of primitive functions including parallel functions such as 'por'.] Hence the non-standard semantics of the function f with $f = \lambda e$ may be defined by $(\boldsymbol{\lambda} v.\mathcal{E}[\![e]\!]\,\langle\langle\rangle, v\rangle)^\sharp$.

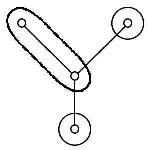
In the sections following we will develop a constructive variant of this definition which does not resort to the standard semantics. Unfortunately we have to make one further restriction as shown in the next section which basically excludes parallel primitive functions, since parallelism spoils compositionality of abstractions.

5.2 Least abstractions of stable functions

One of the central properties of denotational semantics is compositionality: The meaning of a compound object is determined by the meaning of its subcomponents. With respect to the non-standard semantics one would like $(\varphi \circ \psi)^\sharp = \psi^\sharp \circ \varphi^\sharp$ for $\psi : D \to E$ and $\varphi : E \to F$. However, a counterexample is easily constructed. Consider por $\circ\, \psi$ with $\psi : (\mathbb{B}_\bot)_\bot \circ\!\!\to (\mathbb{B}_\bot)_\bot \otimes (\mathbb{B}_\bot)_\bot$ given by $\psi(v) = \langle \textbf{up}\,\textbf{t}, v\rangle$.



por $\circ\, \psi$



$\pi_{\text{por}} \circ \psi$

Since por $\circ\, \psi$ equals $(\boldsymbol{\lambda} v.\textbf{t})^\dagger$ we have $(\text{por} \circ \psi)^\sharp(\top\!\!\top) = (\bot\!\!\bot \oplus \bot\!\!\bot)_\bot$. But on the other hand $(\psi^\sharp \circ \text{por}^\sharp)(\top\!\!\top) = \psi^\sharp(\pi_{\text{por}}) = (\bot\!\!\bot \oplus \top\!\!\top)_\bot$. This failure is caused by 'parallel or': Since $M(\text{por}, (\textbf{up}\,\textbf{t}, \textbf{up}\,\textbf{t}), \textbf{t})$ contains the two elements $(\textbf{up}\,\textbf{t}, \textbf{up}\,\bot)$ and $(\textbf{up}\,\bot, \textbf{up}\,\textbf{t})$, the projection $\text{por}^\sharp(\top\!\!\top)$ leaves the point $(\textbf{up}\,\textbf{t}, \textbf{up}\,\textbf{t})$ unchanged. However, ψ only reaches $(\textbf{up}\,\textbf{t}, \textbf{up}\,\bot)$ and $(\textbf{up}\,\textbf{t}, \textbf{up}\,\textbf{t})$, so $(\textbf{up}\,\textbf{t}, \textbf{up}\,\textbf{t})$ may be lowered to $(\textbf{up}\,\textbf{t}, \textbf{up}\,\bot)$. The problem disappears if the sets $\{v \sqsubseteq x \mid y \sqsubseteq \varphi(v)\}$ always contain a *least* element.

5.6 Definition *A function* $\varphi : D \to E$ *is called* stable *iff* $\{v \sqsubseteq x \mid y \sqsubseteq \varphi(v)\}$ *has a least element, for all* $x \in D$ *and* $y \in E$*, such that* $y \sqsubseteq \varphi\, x$*. The least element is written as* $m(\varphi, x, y)$*. The set of all stable functions from* D *to* E *is denoted by* $D \to_s E$*. The stable order* $\sqsubseteq_s$ *is defined by, for stable* $\varphi_1, \varphi_2 : D \to_s E$*,*

$$\begin{aligned} \varphi_1 \sqsubseteq_s \varphi_2 \iff\ & \varphi_1 \sqsubseteq \varphi_2 \wedge (\forall x \in D)(\forall y \in E) \\ & y \sqsubseteq \varphi_1(x) \Longrightarrow m(\varphi_1, x, y) = m(\varphi_2, x, y). \end{aligned}$$

The theory of stability was introduced by Berry [20] in an attempt to extend the property of sequentiality respectively approximations thereof to higher types.

The value $m(\varphi, x, y)$ may be paraphrased as the *definite* information needed from x to reach y by applying φ. The 'sequential or' is an example for a stable function, while 'parallel or' serves as a counterexample, since there is no least value $v \sqsubseteq (\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\mathbf{t})$ such that $\mathbf{t} \sqsubseteq \mathrm{por}(v)$. To see that stability is in fact an approximation to sequentiality consider the least monotonic function $h : (\mathbb{B}_\perp)_\perp \otimes (\mathbb{B}_\perp)_\perp \otimes (\mathbb{B}_\perp)_\perp \circ\!\!\to \mathbb{B}_\perp$ such that

$$h(\mathbf{up}\,\mathbf{t}, \mathbf{up}\,\mathbf{f}, \mathbf{up}\,\perp) = h(\mathbf{up}\,\mathbf{f}, \mathbf{up}\,\perp, \mathbf{up}\,\mathbf{t}) = h(\mathbf{up}\,\perp, \mathbf{up}\,\mathbf{t}, \mathbf{up}\,\mathbf{f}) = \mathbf{t}.$$

Berry explicates the difference between stability and sequentiality as follows.

> "On a computational point of view, stability implies the existence of 'optimal computations' but not the existence of 'optimal computation rules' which would correspond to sequentiality."[20]

The stable ordering of functions is tailored in some sense towards their computational interpretation. For stable φ_1 and φ_2, $\varphi_1 \sqsubseteq_s \varphi_2$ means that not only the values of φ_1 approximate those of φ_2 but also the computations involved in their calculation. If $\varphi_1(x)$ reaches y, then the minimal information must be equal in both cases. For $\varphi_1 \sqsubseteq \varphi_2$ only $m(\varphi_1, x, y) \sqsupseteq m(\varphi_2, x, y)$ holds, that is, φ_2 requires less information to reach y.

The following corollary adapts Theorem 5.3 to stable functions. [Davis derives a similar result in [47]. While he does not require Axiom I he only shows that there is a least projection not necessarily being finitary which satisfies the safety condition.]

5.7 Corollary *Let D be an I-domain and let $\varphi : D \circ\!\!\to E$ be a stable function. Then $\varphi^\sharp$ with*

$$\varphi^\sharp(\pi)(v) \quad = \quad m(\varphi, v, \pi(\varphi\, v))$$

is the least abstraction of φ.

Proof It remains to prove that the two characterizations of φ coincide. Let

$$M = \{\, m(\varphi, d, \pi(\varphi\, d)) \mid d \in F(D) \cap {\downarrow} v \,\},$$

we show that $m(\varphi, v, \pi(\varphi\, v)) = \bigsqcup M$. Since M is bounded by v we have $\bigsqcup M \sqsubseteq v$. Moreover,

$$\pi(\varphi\, v) = \bigsqcup \pi(\varphi(F(D) \cap {\downarrow} v)) \sqsubseteq \bigsqcup \varphi(M) \sqsubseteq \varphi(\bigsqcup M).$$

Finally, assume that $u \sqsubseteq v$ and $\pi(\varphi\, v) \sqsubseteq \varphi(u)$. Since

$$m(\varphi, d, \pi(\varphi\, d)) \sqsubseteq m(\varphi, v, \pi(\varphi\, v)) \sqsubseteq u,$$

for $d \in F(D) \cap {\downarrow} v$, M is bounded by u and consequently $\bigsqcup M \sqsubseteq u$. ∎

The lemma below shows that least abstractions of stable functions are in fact compositional.

5.8 Lemma *Let $\psi : D \to E$ and $\varphi : E \to F$ be stable functions. Then*

$$(\varphi \circ \psi)^\sharp \quad = \quad \psi^\sharp \circ \varphi^\sharp.$$

Proof We employ the characterization of the least abstraction given in Corollary 5.7.

$$\begin{aligned}
(\varphi \circ \psi)^\sharp(\pi)(v) &= m(\varphi \circ \psi, v, \pi((\varphi \circ \psi)(v))) && \text{by Corollary 5.7} \\
&= m(\psi, v, m(\varphi, \psi\, v, \pi(\varphi(\psi\, v)))) && \text{by (A.4)} \\
&= m(\psi, v, \varphi^\sharp(\pi)(\psi\, v)) && \text{by Corollary 5.7} \\
&= \psi^\sharp(\varphi^\sharp(\pi))(v) && \text{by Corollary 5.7}
\end{aligned}$$ ∎

5.3 Least abstractions and lifted domains

This section considers the programming language constructs `freeze` and `unfreeze` and derives least abstractions for their semantic counterparts.

Let us start with the simpler one. The standard semantics maps `unfreeze` to the function **down**. As **down** has type $D_\perp \circ\!\!\to D$ its least abstraction must be of type $\|D\| \circ\!\!\to \|D_\perp\|$. Remember that `unfreeze` triggers the evaluation of its argument. Hence its least abstraction equals the strict lift $(\cdot)_{\oplus}$.

$$\mathbf{down}^\sharp(\pi) \quad = \quad \pi_{\oplus}$$

The semantic counterpart of `freeze` is the function **up**. Contrary to **down** we are forced to take the context in which **up** is situated into account, since **up** is the sole non-strict functions in an otherwise strict setting. The standard semantics interprets an expression e as a *strict* mapping from the value environment to a value: **ValEnv** $\circ\!\!\to$ **Val** (we may safely assume that the function environment is fixed). Consequently the expression `freeze` e is of type **ValEnv** $\circ\!\!\to$ **Val**$_\perp$. Note that the mapping is still strict. This perfectly models the behaviour of an actual implementation. If the construction of the run-time environment fails, for example, due to lack of memory, the building of the closure incorporating the environment cannot succeed. Thus if φ is the meaning of e, `freeze` e denotes $\mathbf{strict}(\mathbf{up} \circ \varphi)$. Hence we are looking for the least abstraction of $\psi = \mathbf{strict}(\mathbf{up} \circ \varphi)$. Recall from the introduction that the unlifted demand on ψ is passed to φ, whose result is adjusted according to the lift of the demand (the adjustment is termed the *guard operation*). Consider $\varphi =$ not as a simple example. The least abstraction of 'not' is

$$\text{not}^\sharp(\pi_1 \oplus \pi_2) \quad = \quad (\pi_2 \oplus \pi_1)_{\oplus}.$$

The strict lift indicates that 'not' is strict irrespective of the context. Assume that the demand on ψ is $[\mathbf{f}]_{\oplus}$ which specifies that the result is definitely needed. We calculate not$^\sharp$ $[\mathbf{f}]$ yielding $[\mathbf{t}]_{\oplus}$. Since the lift of the demand on ψ is strict, $[\mathbf{t}]_{\oplus}$ also represents the demand on ψ's argument. Conversely, if the demand equals $[\mathbf{f}]_\perp$ indicating that the result may or may not be required, the resulting projection must be weakened to $[\mathbf{t}]_\perp$. In this case the guard operator may be paraphrased as 'if the result is not required neither is the argument'. Formally, the guard operation is defined by

$$\begin{aligned} \oplus \rhd \pi \quad &= \quad \pi \\ \perp \rhd \pi \quad &= \quad \pi \sqcup \perp\!\!\!\perp_\perp. \end{aligned}$$

Accordingly the least abstraction of ψ is

$$\psi^\sharp(\pi) \quad = \quad \Lambda(\pi) \rhd \varphi^\sharp(\pi\!\downarrow).$$

There is, however, a slight slip in the argument. The guard operation is tailored to lifted domains, that is, φ must be of type $D_\perp \circ\!\!\to E$, while the semantic function $\mathcal{E}[\![e]\!]\ \phi$ is of type **ValEnv** $\circ\!\!\to$ **Val** (note that **ValEnv** is not isomorphic to $D_\perp$). The situation is saved if we replace $\perp\!\!\!\perp_\perp$ by the unique element $1 \in \|\mathbf{ValEnv}\|$ which is neutral with respect to $\&$. The conjunctive unit is the least projection which maps only $\perp$ to $\perp$, that is, which is bottom-reflecting. To see why this is in fact the right choice consider $\psi^\sharp(\perp\!\!\!\perp_\perp)$. First of all, we have

$$\psi^\sharp(\perp\!\!\!\perp_\perp) = \perp \rhd \varphi^\sharp(\perp\!\!\!\perp) = \perp \rhd \perp\!\!\!\perp = 1.$$

The demand $\perp\!\!\!\perp_\perp$ only requires the result to be proper. Thus $\psi^\sharp(\perp\!\!\!\perp_\perp)$ may not send any proper values to $\perp$ and must be as small as possible.

The following lemma shows that our standard set of domain operators allows for conjunctive units.

5.9 Lemma *The conjunctive units on $D_\perp$, $D_1 \oplus \cdots \oplus D_n$, and $D_1 \otimes \cdots \otimes D_n$ satisfy*

1. $1_{D_\perp} \;=\; \amalg_\perp$
2. $1_{D_1 \oplus \cdots \oplus D_n} \;=\; 1_{D_1} \oplus \cdots \oplus 1_{D_n}$
3. $1_{D_1 \otimes \cdots \otimes D_n} \;=\; 1_{D_1} \otimes \cdots \otimes 1_{D_n}$.

Proof Ad 1) Cf to Lemma 4.23. Ad 2)

$$
\begin{array}{rll}
 & (1_{D_1} \oplus \cdots \oplus 1_{D_n}) \mathbin{\&} \pi & \\
= & (1_{D_1} \oplus \cdots \oplus 1_{D_n}) \mathbin{\&} (\pi \downarrow 1 \oplus \cdots \oplus \pi \downarrow n) & \text{Lemma 4.11} \\
= & (1_{D_1} \mathbin{\&} \pi \downarrow 1) \oplus \cdots \oplus (1_{D_n} \mathbin{\&} \pi \downarrow n) & \text{Lemma 4.25} \\
= & \pi \downarrow 1 \oplus \cdots \oplus \pi \downarrow n & 1_{D_i} \mathbin{\&} \pi \downarrow i = \pi \downarrow i \\
= & \pi & \text{Lemma 4.11}
\end{array}
$$

Ad 3) The proof basically employs Corollary 4.16 which shows that an arbitrary projection on a smash product may be factored into a set of independent projections.

$$
\begin{array}{rll}
 & (1_{D_1} \otimes \cdots \otimes 1_{D_n}) \mathbin{\&} \pi & \\
= & (1_{D_1} \otimes \cdots \otimes 1_{D_n}) \mathbin{\&} \bigsqcup\{\, \pi_1 \otimes \cdots \otimes \pi_n \mid \pi_1 \otimes \cdots \otimes \pi_n \sqsubseteq \pi \,\} & \text{Corollary 4.16} \\
= & \bigsqcup\{\, (1_{D_1} \otimes \cdots \otimes 1_{D_n}) \mathbin{\&} (\pi_1 \otimes \cdots \otimes \pi_n) \mid \pi_1 \otimes \cdots \otimes \pi_n \sqsubseteq \pi \,\} & \\
 & & \lfloor\text{Join-Infinite Distributive Law} \\
= & \bigsqcup\{\, (1_{D_1} \mathbin{\&} \pi_1) \otimes \cdots \otimes (1_{D_n} \mathbin{\&} \pi_n) \mid \pi_1 \otimes \cdots \otimes \pi_n \sqsubseteq \pi \,\} & \text{Lemma 4.25} \\
= & \bigsqcup\{\, \pi_1 \otimes \cdots \otimes \pi_n \mid \pi_1 \otimes \cdots \otimes \pi_n \sqsubseteq \pi \,\} & 1_{D_i} \mathbin{\&} \pi_i = \pi_i \\
= & \pi & \text{Corollary 4.16} \blacksquare
\end{array}
$$

One can show, in fact, that $\|D\|$ contains a conjunctive unit if and only if for each $v^\circ \in D^\circ$ there is a *unique* element $a \in \mathcal{A}(D)$ such that $a \sqsubseteq v^\circ$. Consequently $\|\mathbf{2} \times \mathbf{2}\|$ contains no conjunctive unit, since $\mathcal{A}(\mathbf{2} \times \mathbf{2}) \cap {\downarrow}(1,1) = \{(0,1),(1,0)\}$.

5.10 Definition *Let $\pi \in \|D\|$. The* guard operator *$\ell \rhd \pi$ with $\ell \in \{\oplus, \perp\}$ is defined by*

$$
\begin{array}{rcl}
\oplus \rhd \pi & = & \pi \\
\perp \rhd \pi & = & \pi \sqcup 1_D .
\end{array}
$$

5.11 Lemma *Let D and E be I-domains and let $\varphi : D \circ\!\!\rightarrow_s E$ be a stable function. Then*

1. $(\mathbf{strict}(\mathbf{up} \circ \varphi))^\sharp(\pi) \;=\; \Lambda(\pi) \rhd \varphi^\sharp(\pi{\downarrow})$,
2. $\mathbf{down}^\sharp(\pi) \;=\; \pi_{\oplus}$.

Proof Ad 1) Let $\psi = \mathbf{strict}(\mathbf{up} \circ \varphi)$, then

$$
\begin{array}{rcll}
m(\psi, x, \mathbf{up}\perp) & = & 1_D(x) & (5.1) \\
m(\psi, x, \mathbf{up}\, v^\circ) & = & m(\varphi, x, v^\circ) & (5.2)
\end{array}
$$

Notice that $m(\varphi, x, y) = \perp$ iff φ is strict. Case $\pi_{\oplus}$:

$$
\begin{array}{rcll}
\psi^\sharp(\pi_{\oplus})(v) & = & m(\psi, v, \pi_{\oplus}(\psi\, v)) & \text{by Corollary 5.7} \\
 & = & m(\varphi, v, \mathbf{down}(\pi_{\oplus}(\psi\, v))) & \text{by (5.2), since } \mathbf{up}\perp \notin \mathbf{im}\, \pi_{\oplus} \\
 & = & m(\varphi, v, \pi(\varphi\, v)) & \text{def. } \pi_{\oplus} \\
 & = & \varphi^\sharp(\pi)(v) & \text{by Corollary 5.7}
\end{array}
$$

Case $\pi_\perp$: First note that $\psi^\sharp(\perp\!\!\!\perp_\perp)(v^\circ) = m(\psi, v^\circ, \perp\!\!\!\perp_\perp(\psi\, v^\circ)) = m(\psi, v^\circ, \textbf{up}\,\perp) = 1_D(v^\circ)$ by (5.1), for $v^\circ \in D^\circ$. Hence $\psi^\sharp(\perp\!\!\!\perp_\perp) = 1_D$.

$$\begin{aligned}
\psi^\sharp(\pi_\perp) &= \psi^\sharp(\pi_\oplus \sqcup \perp\!\!\!\perp_\perp) && \text{by Lemma 4.25}\\
&= \psi^\sharp(\pi_\oplus) \sqcup \psi^\sharp(\perp\!\!\!\perp_\perp) && \text{by Lemma 5.4}\\
&= \varphi^\sharp(\pi) \sqcup 1_D && \text{see above}
\end{aligned}$$

Ad 2) Again, we start with a characterization of 'm'.

$$m(\textbf{down}, x, y) \quad = \quad \textbf{strict up}\; y \tag{5.3}$$

The proposition follows by a simple calculation.

$$\begin{aligned}
\textbf{down}^\sharp(\pi)(v) &= m(\textbf{down}, v, \pi(\textbf{down}\; v)) && \text{by Corollary 5.7}\\
&= (\textbf{strict up} \circ \pi \circ \textbf{down})(v) && \text{by (5.3)}\\
&= \pi_\oplus(v) && \text{def. } \pi_\oplus \blacksquare
\end{aligned}$$

5.4 Least abstractions and smash products

The de Bruijn transformation introduced in Section 2.4 clearly reveals the nature of variables. A variable boils down to a projection function[1] which takes a run-time environment and accesses the corresponding entry. Run-time environments in turn are nested pairs of tuples. This concretizes their usual representation as (finite) functions from variables to values. Thus apart from their explicit use in sequences tuples and projection functions are implicitly employed in the maintenance of the run-time environment.

Projection functions forget information, $\textbf{on}_i$, for example, accesses the i-th component of a tuple while ignoring the remaining $n-1$ components. Speaking in terms of requirements the demand on the remaining components is rather low. Since we work with smash products they must in fact only be proper. This motivates the following definition.

5.12 Definition *Let π be a projection on D_i. The projection $(i\colon \pi)$ on $D_1 \otimes \cdots \otimes D_n$ is defined as follows.*

$$(i\colon \pi) \quad = \quad \pi_1 \otimes \cdots \otimes \pi_n \; \textbf{where}\; \pi_j = \begin{cases} \pi & \textbf{if}\; i = j \\ 1_{D_j} & \textbf{otherwise}. \end{cases}$$

Since $(i\colon \pi)$ consists primarily of conjunctive units it interacts nicely with the conjunction of demands: $(1\colon \pi_1) \mathbin{\&} \cdots \mathbin{\&} (n\colon \pi_n) = \pi_1 \otimes \cdots \otimes \pi_n$.

Turning to the construction of smash products it is profitable to define a functional variant of tupling: $\langle \varphi_1, \ldots, \varphi_n\rangle(v) = \langle \varphi_1\, v, \ldots, \varphi_n\, v\rangle$. The evaluation of the sequence $e_1 \ldots e_n$ causes the evaluation of all its subexpressions implying a conjunctive demand on their definedness. The operation $\&$ was introduced to model this computational behaviour. Let us, for the moment, restrict ourselves to independent projections. In this special case the least abstraction of $\langle \varphi_1, \ldots, \varphi_n\rangle$ is given by

$$\langle \varphi_1, \ldots, \varphi_n\rangle^\sharp(\pi_1 \otimes \cdots \otimes \pi_n) \quad = \quad \varphi_1^\sharp(\pi_1) \mathbin{\&} \cdots \mathbin{\&} \varphi_n^\sharp(\pi_n).$$

[1] Do not confuse projection function with projection: $\textbf{on}_i$ is the prototypical projection function, while the term projection is used solely in the sense of Definition 4.1.

Recall that a value is acceptable to $\pi_1 \& \pi_2$ if and only if it is acceptable to both π_1 and π_2 which characterizes precisely the behaviour of smash products. The generalization of $\langle\varphi_1, \ldots, \varphi_n\rangle^\sharp$ to arbitrary projections on smash products poses no problems. Central to this undertaking are Corollary 4.16, which implies that a projection on $D_1 \oplus \cdots \oplus D_n$ is equal to the join of its independent approximations, and Lemma 5.4, which shows that least abstractions preserve joins.

5.13 Lemma *Let D and E_i be I-domains and let $\varphi_i : D \circ\!\!\rightarrow E_i$ be stable functions. Then*

1. $\mathsf{on}_i^\sharp(\pi) \;=\; (i\colon \pi)$

2. $\langle\varphi_1, \ldots, \varphi_n\rangle^\sharp(\pi) \;=\; \bigsqcup\{\, \varphi_1^\sharp(\pi_1) \,\&\, \cdots \,\&\, \varphi_n^\sharp(\pi_n) \mid \pi_1 \otimes \cdots \otimes \pi_n \sqsubseteq \pi \,\}$

Proof Ad 1) As usual we resort to Corollary 5.7. Using

$$m(\mathsf{on}_i, \bot, y) \;=\; \bot \tag{5.4}$$

$$m(\mathsf{on}_i, (x_1, \ldots, x_n), y) \;=\; \langle 1_{D_1}\, x_1, \ldots, y, \ldots, 1_{D_n}\, x_n\rangle \tag{5.5}$$

we get

$$\begin{aligned}
\mathsf{on}_i^\sharp(\pi)(v_1, \ldots, v_n) &= m(\mathsf{on}_i, (v_1, \ldots, v_n), \pi\, v_i) && \text{by Corollary 5.7}\\
&= \langle 1_{D_1}\, v_1, \ldots, \pi\, v_i, \ldots, 1_{D_n}\, v_n\rangle && \text{by (5.4)}\\
&= (i\colon \pi)(v_1, \ldots, v_n) && \text{def. } (i\colon \pi)
\end{aligned}$$

Ad 2) Let $\varphi = \langle\varphi_1, \ldots, \varphi_n\rangle$. We start by proving $\varphi^\sharp(\pi_1 \otimes \cdots \otimes \pi_n) = \varphi_1^\sharp(\pi_1) \& \cdots \& \varphi_n^\sharp(\pi_n)$. Remember that $\&$ is the strict variant of $\sqcup$.

$$m(\varphi, x, \langle y_1, \ldots, y_n\rangle) \;=\; m(\varphi_1, x, y_1) \,\&\, \cdots \,\&\, m(\varphi_1, x, y_1) \tag{5.6}$$

Hence,

$$\begin{aligned}
&\varphi^\sharp(\pi_1 \otimes \cdots \otimes \pi_n)(v)\\
=\;& m(\varphi, v, (\pi_1 \otimes \cdots \otimes \pi_n)(\varphi\, v)) && \text{Corollary 5.7}\\
=\;& m(\varphi, v, \langle \pi_1(\varphi_1\, v), \ldots, \pi_n(\varphi_n\, v)\rangle) && \text{def. } \pi_1 \otimes \cdots \otimes \pi_n\\
=\;& m(\varphi_1, v, \pi_1(\varphi_1\, v)) \,\&\, \cdots \,\&\, m(\varphi_n, v, \pi_n(\varphi_n\, v)) && \text{by (5.6)}\\
=\;& \varphi_1^\sharp(\pi_1)(v) \,\&\, \cdots \,\&\, \varphi_n^\sharp(\pi_n)(v) && \text{Corollary 5.7}\\
=\;& (\varphi_1^\sharp(\pi_1) \,\&\, \cdots \,\&\, \varphi_n^\sharp(\pi_n))(v)
\end{aligned}$$

Corollary 4.16 implies that each projection on a smash product equals the join of its independent approximations.

$$\begin{aligned}
\varphi^\sharp(\pi) &= \varphi^\sharp(\bigsqcup\{\, \pi_1 \otimes \cdots \otimes \pi_n \mid \pi_1 \otimes \cdots \otimes \pi_n \sqsubseteq \pi \,\}) && \text{by Corollary 4.16}\\
&= \bigsqcup\{\, \varphi^\sharp(\pi_1 \otimes \cdots \otimes \pi_n) \mid \pi_1 \otimes \cdots \otimes \pi_n \sqsubseteq \pi \,\}) && \text{by Lemma 5.4}\\
&= \bigsqcup\{\, \varphi_1^\sharp(\pi_1) \,\&\, \cdots \,\&\, \varphi_n^\sharp(\pi_n) \mid \pi_1 \otimes \cdots \otimes \pi_n \sqsubseteq \pi \,\}) && \text{see above} \blacksquare
\end{aligned}$$

5.5 Least abstractions and coalesced sums

Sum domains model data types typically found in functional languages. Elements of a data type are build by means of constructor functions such as `[]` or ‘`:`’. The least abstraction of their semantical counterpart, the injection function $\mathbf{in}_i$, is particularly easy to construct. A demand on the result of $\mathbf{in}_i$ is of the form $\pi_1 \oplus \cdots \oplus \pi_n$, the components specifying the demands on the summands of the sum domain. Hence the least abstraction of $\mathbf{in}_i$ is given by

$$\mathbf{in}_i^\sharp(\pi) \quad = \quad \pi \downarrow i.$$

The $\underline{\texttt{case}}$-construct is by far the most complicated programming language construct to analyse. To reduce its complexity we factor a single $\underline{\texttt{case}}$-expression involving n branches into n $\underline{\texttt{case}}$-expressions each having only a single branch. The idea is in fact borrowed from work on the compilation of pattern-matching, cf [130].

$$\begin{array}{l} \underline{\texttt{case}}\; e\; \underline{\texttt{of}}\; c_1 \rightarrow e_1 \mid \cdots \mid c_n \rightarrow e_n \\ \cong \quad \underline{\texttt{case}}\; e\; \underline{\texttt{of}}\; c_1 \rightarrow e_1 \;?\; \cdots \;?\; \underline{\texttt{case}}\; e\; \underline{\texttt{of}}\; c_n \rightarrow e_n \end{array} \tag{5.7}$$

The operational behaviour of ‘?’ is as follows: If the pattern match in the first argument succeeds, the value of the branch is returned. Otherwise, the value of the second argument is returned. Since we model run-time errors such as ‘missing case’ by the bottom element, ‘?’ respectively its semantic counterpart $\boxplus$ may be characterized by

$$\begin{array}{rcl} \bot \boxplus x &=& x \\ x \boxplus \bot &=& x. \end{array}$$

Technical remark: The operator $\boxplus$ equals the join $\sqcup$ with the restriction that *at most one* of its components may be proper. This restriction is necessary to guarantee the correctness of the program transformation above. The point is that $\boxplus$ cannot distinguish between failure of a match and non-termination of a branch. Consequently we must ensure that at most one of the embedded $\underline{\texttt{case}}$-constructs succeeds, which in turn is granted by the syntax, since the constructor names are pairwise different.

The syntactical transformation is mirrored in the defining equations of the standard semantics.

$$\begin{array}{rcl} \mathcal{E}[\![\underline{\texttt{case}}\; e_1\; \underline{\texttt{of}}\; c \rightarrow e]\!]\; \phi\; \rho &=& \mathcal{E}[\![e]\!]\; \phi\; \langle \rho, \mathbf{out}_c(\mathcal{E}[\![e_1]\!]\; \phi\; \rho)\rangle \\ \mathcal{E}[\![e_1 \;?\; e_2]\!]\; \phi\; \rho &=& \mathcal{E}[\![e_1]\!]\; \phi\; \rho \boxplus \mathcal{E}[\![e_2]\!]\; \phi\; \rho \end{array}$$

Now, life is more comfortable, since it suffices to derive least abstractions for $\mathbf{out}_c$ and $\boxplus$, respectively. The result of $\mathbf{out}_c$ is proper if and only if its argument is of the form $\mathbf{in}_c\; v$. Hence $\mathbf{out}_c^\sharp(\pi)$ sets elements of the form $\mathbf{in}_c\; v$ to $\mathbf{in}_c(\pi\; v)$ and to $\bot$ otherwise. This motivates the following definition which is dual to Definition 5.12.

5.14 Definition *Let π be a projection on D_i. The projection $[i\colon \pi]$ on $D_1 \oplus \cdots \oplus D_n$ is defined as follows.*

$$[i\colon \pi] \quad = \quad \pi_1 \oplus \cdots \oplus \pi_n \;\textbf{where}\; \pi_j = \begin{cases} \pi & \textbf{if } i = j \\ \perp\!\!\!\perp & \textbf{otherwise} \end{cases}$$

Note that $[i\colon \pi]$ could have been equally defined by $[i\colon \pi] = \mathbf{in}_i \circ \pi \circ \mathbf{out}_i$. Since $[i\colon \pi]$ consists primarily of disjunctive units it interacts nicely with the disjunction of demands: $[1\colon \pi_1] \sqcup \cdots \sqcup [n\colon \pi_n] = \pi_1 \oplus \cdots \oplus \pi_n$.

Now, turning to $\boxplus$ we first map the operator to functions by setting $(\varphi_1 \boxplus \cdots \boxplus \varphi_n)(v) = \varphi_1(v) \boxplus \cdots \boxplus \varphi_n(v)$. Since $\boxplus$ is a restricted form of the join, it comes as no surprise that its least abstraction equals the disjunction of the least abstractions of $\varphi_1, \ldots, \varphi_n$. The lemma following summarizes the results.

5.15 Lemma *Let D and E be I-domains and let $\varphi_i : D \circ\!\!\rightarrow E$ be stable functions. Then*

1. $\mathsf{in}_c^\sharp(\pi) \quad = \quad \pi \downarrow c$
2. $\mathsf{out}_c^\sharp(\pi) \quad = \quad [c\colon \pi]$
3. $(\varphi_1 \boxplus \cdots \boxplus \varphi_n)^\sharp(\pi) \quad = \quad \varphi_1^\sharp(\pi) \sqcup \cdots \sqcup \varphi_n^\sharp(\pi)$

Proof Ad 1) Using

$$m(\mathsf{in}_c, x, y) \quad = \quad \mathsf{out}_c\, y \tag{5.8}$$

we immediately get

$$\begin{aligned} \mathsf{in}_c^\sharp(\pi)(v) &= m(\mathsf{in}_c, v, \pi(\mathsf{in}_c\, v)) && \text{by Corollary 5.7} \\ &= (\mathsf{out}_c \circ \pi \circ \mathsf{in}_c)(v) && \text{by (5.8)} \\ &= (\pi \downarrow c)(v) && \text{def. } \pi \downarrow c \end{aligned}$$

Ad 2) Conversely,

$$m(\mathsf{out}_c, x, y) \quad = \quad \mathsf{in}_c\, y \tag{5.9}$$

implies

$$\begin{aligned} \mathsf{out}_c^\sharp(\pi)(v) &= m(\mathsf{out}_c, v, \pi(\mathsf{out}_c\, v)) && \text{by Corollary 5.7} \\ &= (\mathsf{in}_c \circ \pi \circ \mathsf{out}_c)(v) && \text{by (5.9)} \\ &= [c\colon \pi](v) && \text{def. } [c\colon \pi] \end{aligned}$$

Ad 3) Recall that the use of $\boxplus$ requires that for all $v \in D$ there is at most one i such that $\varphi_i\, v \neq \bot$. Hence the set $S = \{\varphi_i\, v\}_{1 \leqslant i \leqslant n}$ is directed and we have,

$$\begin{aligned} & (\varphi_1 \boxplus \cdots \boxplus \varphi_n)^\sharp(\pi)(v) \\ = \; & m(\varphi_1 \boxplus \cdots \boxplus \varphi_n, v, \pi(\varphi_1\, v \boxplus \cdots \boxplus \varphi_n\, v)) && \text{by Corollary 5.7} \\ = \; & m(\varphi_1 \boxplus \cdots \boxplus \varphi_n, v, \pi(\varphi_1\, v) \boxplus \cdots \boxplus \pi(\varphi_n\, v)) && S \text{ is directed} \\ = \; & m(\varphi_1, v, \pi(\varphi_1\, v)) \sqcup \cdots \sqcup m(\varphi_n, v, \pi(\varphi_n\, v)) && \text{as } (\exists_{\leqslant 1} i)\, \varphi_i(v) \neq \bot \\ = \; & \varphi_1^\sharp(\pi)(v) \sqcup \cdots \sqcup \varphi_n^\sharp(\pi)(v) && \text{by Corollary 5.7} \; \blacksquare \end{aligned}$$

5.6 Least abstractions and least fixpoints

The meaning of recursive value and function definitions is given by least fixpoints of corresponding functionals. First notice that in both cases the functionals involve higher types. This is quite obvious in the case of function definitions—if the fixpoint is of type $D \to E$ the associated functional has the type $(D \to E) \to (D \to E)$. Despite appearance value definitions are not radically different since expressions denote functions mapping value environments to values. Consequently the functional associated with a value definition is of type $(\mathbf{ValEnv} \circ\!\!\to \mathbf{Val}) \to (\mathbf{ValEnv} \circ\!\!\to \mathbf{Val})$.

The fact that parts of the semantics involve higher-order functions forces us to generalize the theory developed to higher types. Let us first clarify what the non-standard semantics of such functionals should be. The abstraction of $\psi : (D \to E) \to (D \to E)$ is certainly *not* a projection transformer which maps a demand on the resulting function to a demand on the function passed as an argument. A function specifies an action corresponding in an actual implementation to a sequence of machine instructions. Hence it makes no sense to define projections (which specify a certain degree of evaluation) on function spaces. To see what an abstraction might look like, assume that we want to infer the least abstraction of $\psi(\varphi)$, for a given function $\varphi : D \to E$. Knowing the least abstraction of φ we may proceed as in the first-order case. Hence the natural non-standard semantics of ψ is a functional which maps abstractions specifying the behaviour of ψ's argument to abstractions characterizing the result. Notice that the order of argumentation is forward from the argument to the result whereas the natural order of analysis in the first-order case is backward.

5.16 Definition *A function* $\Psi : (\|E\| \to \|D\|) \to (\|G\| \to \|F\|)$ *is called projection-transformer transformer. Let* $\psi : (D \to E) \to (F \to G)$, *then* Ψ *is an* abstraction of ψ *iff for all* $\varphi : D \to E$ *and* $\tau : \|E\| \to \|D\|$ *the following holds: if* τ *is an abstraction of* φ, *then* $\Psi(\tau)$ *is an abstraction of* $\psi(\varphi)$.

The *least* abstraction of a functional $\psi : (D \to E) \to (F \to G)$ may then be defined as a projection-transformer transformer mapping least abstractions to least abstractions.

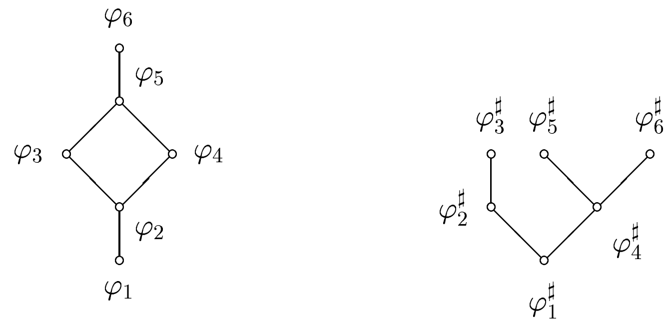
5.17 Definition *Let* $E \Rightarrow D = \{\, \varphi^\sharp \mid \varphi \in D \circ\!\!\to E \,\}$ *and let* $\psi : (D \circ\!\!\to E) \to (F \circ\!\!\to G)$. *The* least abstraction of ψ *is the unique function* $\Psi : (E \Rightarrow D) \to (G \Rightarrow F)$ *which makes the following diagram commute.*

$$\begin{array}{ccc} D \circ\!\!\to E & \xrightarrow{\psi} & F \circ\!\!\to G \\ {\scriptstyle (\cdot)^\sharp}\downarrow & & \downarrow{\scriptstyle (\cdot)^\sharp} \\ E \Rightarrow D & \xrightarrow[\Psi]{} & G \Rightarrow F \end{array}$$

The function Ψ is uniquely determined because the map $(\cdot)^\sharp$ is one-to-one by Theorem 5.5. As usual we denote the least abstraction of the functional ψ by $\psi^\sharp$.

If φ is defined as the least fixpoint of the functional $\psi : (D \to E) \to (D \to E)$, the least abstraction of φ is, in fact, given by the least fixpoint of the functional $\psi^\sharp$.[2] The proof involves a simple Fixpoint Induction. The main difficulty is to show that the predicate P with $P(\varphi, \tau) \iff \varphi^\sharp = \tau$ which is employed in the proof is admissible. This would be granted if $(\cdot)^\sharp$ considered as a mapping of type $(D \to E) \to (E \Rightarrow D)$ were continuous. However, $(\cdot)^\sharp$ is not even monotone as the following example shows. The picture below displays the strict, monotonic functions on **3** and their abstractions under the usual function ordering.

[2]We must restrict ourselves to stable functions, see below.



We have $\varphi_2 \sqsubseteq \varphi_4$, but $\varphi_2^\sharp \not\sqsubseteq \varphi_4^\sharp$. Davis [47] shows that this 'defect' may be repaired if we restrict ourselves to stable functions and use the stable function ordering on the argument domain of $(\cdot)^\sharp$. [Davis proves that $(\cdot)^\sharp$ is monotone wrt $\sqsubseteq_s$.]

5.18 Lemma *Let D be an I-domain. The map $(\cdot)^\sharp : (D \multimap_s E) \to (E \Rightarrow D)$ is continuous with respect to $\sqsubseteq_s$.*

Proof Let $\Phi \subseteq D \multimap_s E$ be directed wrt $\sqsubseteq_s$. Using

$$m(\bigsqcup \Phi, x, y) \quad = \quad \bigsqcup\{\, m(\varphi, x, y \sqcap \varphi(x)) \mid \varphi \in \Phi \,\} \tag{5.10}$$

we get

$$\begin{array}{rcll}
(\bigsqcup \Phi)^\sharp(\pi)(v) & = & m(\bigsqcup \Phi, v, \pi((\bigsqcup \Phi)(v))) & \text{by Corollary 5.7} \\
& = & \bigsqcup\{\, m(\varphi, v, \pi((\bigsqcup \Phi)(v)) \sqcap \varphi(v)) \mid \varphi \in \Phi \,\} & \text{by (5.10)} \\
& \sqsupseteq & \bigsqcup\{\, m(\varphi, v, \pi(\varphi(v))) \mid \varphi \in \Phi \,\} & \\
& = & \bigsqcup\{\, \varphi^\sharp(\pi)(v) \mid \varphi \in \Phi \,\} & \text{by Corollary 5.7} \\
& = & (\bigsqcup \Phi^\sharp)(\pi)(v) & \text{join is formed pointwise}
\end{array}$$

It remains to show that $\bigsqcup \Phi^\sharp$ satisfies the safety condition.

$$\begin{array}{rcll}
\pi \circ \bigsqcup \Phi & = & \bigsqcup\{\, \pi \circ \varphi \mid \varphi \in \Phi \,\} & \Phi \text{ is directed} \\
& \sqsubseteq & \bigsqcup\{\, \varphi \circ \varphi^\sharp(\pi) \mid \varphi \in \Phi \,\} & \varphi^\sharp \text{ is an abstraction} \\
& \sqsubseteq & \bigsqcup\{\, (\bigsqcup \Phi) \circ \varphi^\sharp(\pi) \mid \varphi \in \Phi \,\} & \text{since } \varphi \sqsubseteq \bigsqcup \Phi \\
& \sqsubseteq & (\bigsqcup \Phi) \circ \bigsqcup\{\, \varphi^\sharp(\pi) \mid \varphi \in \Phi \,\} & \bigsqcup \Phi \text{ is monotone} \\
& = & (\bigsqcup \Phi) \circ (\bigsqcup \Phi^\sharp)(\pi) & \text{join is formed pointwise} \ \blacksquare
\end{array}$$

5.19 Lemma *Let D be an I-domain, let $\psi : (D \multimap_s E) \to (D \multimap_s E)$ be continuous wrt $\sqsubseteq_s$ and let $\varphi_0 : D \multimap_s E$. Then*

$$(\mathsf{lfp}_{\varphi_0}\, \psi)^\sharp \quad = \quad \mathsf{lfp}_{\varphi_0^\sharp}\, \psi^\sharp$$

Proof We show the proposition by Fixpoint Induction. First note that the predicate $P \subseteq (D \multimap_s E) \times (E \Rightarrow D)$ with $P(\varphi, \tau) \iff \varphi^\sharp = \tau$ is admissible, since $(\cdot)^\sharp$ is continuous wrt $\sqsubseteq_s$. Induction basis: Holds trivially. Induction step: Since $\psi^\sharp$ is the least abstraction of ψ, $\varphi^\sharp = \tau$ implies $\psi(\varphi) = \psi^\sharp(\tau)$. $\blacksquare$

meta variable		domain	comment
π	$\in$	$\mathbf{Val}^\sharp$	non-standard values
$\varphi^\sharp$	$\in$	$\mathbf{Fun}^\sharp$	non-standard functions
$\rho^\sharp$	$\in$	$\mathbf{ValEnv}^\sharp$	non-standard value environments
$\phi^\sharp$	$\in$	$\mathbf{FunEnv}^\sharp$	non-standard function environments

Table 5.1: Non-standard semantic domains

5.7 Semantic domains and equations

This section introduces the non-standard semantics bringing together the results of the preceding sections.

As usual we start with the specification of the domains involved. The non-standard domains of the language are shown in Table 5.1. They are defined via domain equations displayed in Table 5.2.

domain equation			comment
$\mathbf{Val}^\sharp$	$=$	$\|\mathbf{Val}\|$	projections
$\mathbf{Fun}^\sharp$	$=$	$\|\mathbf{Val}\| \multimap \|\mathbf{Val}_\perp \otimes \cdots \otimes \mathbf{Val}_\perp\|$	projection transformers
$\mathbf{ValEnv}^\sharp$	$=$	$\|\mathbf{ValEnv}\|$	non-standard value environments
$\mathbf{FunEnv}^\sharp$	$=$	$\mathbf{fun} \to \mathbf{Fun}^\sharp$	non-standard function environments

Table 5.2: Non-standard domain equations

The semantic function is denoted by $\mathcal{E}^\sharp$ indicating that $\mathcal{E}^\sharp[\![e]\!]\,\phi^\sharp$ yields the least abstraction of the function $\mathcal{E}[\![e]\!]\,\phi$. Analogous to the standard semantics the non-standard one is defined with respect to a given set of least abstractions for the primitives. However, as the theory is only applicable to stable functions we are forced the restrict the primitives to sequential functions.

Most of the auxiliary functions employed in the semantic equations have been already introduced in Sections 5.3 to 5.6. The rest is explained in the sequel. The least abstraction of the function $\mathbf{acc}_k$ which accesses the k-th entry in the run-time environment is given by $\langle k\colon\cdot\rangle : \|\mathbf{Val} \otimes \cdots \otimes \mathbf{Val}\| \multimap \|\mathbf{ValEnv}\|$,

$$\begin{aligned} \langle 0\colon \pi\rangle &= (2\colon \pi) \\ \langle k+1\colon \pi\rangle &= (1\colon \langle k\colon \pi\rangle). \end{aligned}$$

If we would use the notation of Lemma 5.13 the semantic equations were likely to become unwieldy. Therefore we adopt the following abbreviation (overloading the operator $\langle\cdot,\ldots,\cdot\rangle$ once again).

$$\langle \tau_1,\ldots,\tau_n\rangle(\pi) = \bigsqcup\{\, \tau_1(\pi_1)\,\&\cdots\&\,\tau_n(\pi_n) \mid \pi_1\otimes\cdots\otimes\pi_n \sqsubseteq \pi \,\}$$

Note that $\langle\rangle(\perp\!\!\!\perp) = \perp\!\!\!\perp$ and $\langle\rangle(\top\!\!\!\top) = 1$ for the special case of $n = 0$. The semantic functions are listed in Table 5.3.

Let $\phi^\sharp$ be a non-standard function environment containing the least abstractions of the functions in ϕ. Then $\mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp$ denotes the least abstraction of $\mathcal{E}[\![e]\!]\ \phi$.

$$
\begin{array}{rcl}
\mathcal{E}^\sharp[\![\mathbf{dbe}]\!] & : & \mathbf{FunEnv}^\sharp \to \mathbf{Val}^\sharp \circ\!\!\rightarrow \mathbf{ValEnv}^\sharp \\
\mathcal{E}^\sharp[\![k.i]\!]\ \phi^\sharp\ \pi & = & \langle k\colon (i\colon \pi)\rangle \\
\mathcal{E}^\sharp[\![\texttt{freeze}\ e]\!]\ \phi^\sharp\ \pi & = & \Lambda(\pi) \rhd \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ (\pi\!\downarrow) \\
\mathcal{E}^\sharp[\![\texttt{unfreeze}\ e]\!]\ \phi^\sharp\ \pi & = & \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ \pi_{\oplus} \\
\mathcal{E}^\sharp[\![s\ e]\!]\ \phi^\sharp\ \pi & = & \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ ([\![s]\!]^\sharp(\pi)) \\
\mathcal{E}^\sharp[\![c\ e]\!]\ \phi^\sharp\ \pi & = & \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ (\pi \downarrow c) \\
\mathcal{E}^\sharp[\![f\ e]\!]\ \phi^\sharp\ \pi & = & \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ (\phi^\sharp(f)(\pi)) \\
\mathcal{E}^\sharp[\![e_1 \dots e_n]\!]\ \phi^\sharp\ \pi & = & \langle \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp, \dots, \mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp\rangle(\pi) \\
\multicolumn{3}{l}{\mathcal{E}^\sharp[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1 \mid \cdots \mid c_n \to e_n]\!]\ \phi^\sharp\ \pi} \\
 & = & \langle \mathsf{id}, \tau_1\rangle(\mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp\ \pi) \sqcup \cdots \sqcup \langle \mathsf{id}, \tau_n\rangle(\mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp\ \pi) \\
 & & \textsf{where}\ \tau_1(\pi) = \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ [c_1\colon \pi] \\
 & & \qquad\quad \dots \\
 & & \qquad\quad \tau_n(\pi) = \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ [c_n\colon \pi] \\
\mathcal{E}^\sharp[\![\underline{\texttt{let}}\ e_1\ \underline{\texttt{in}}\ e]\!]\ \phi^\sharp\ \pi & = & \langle \mathsf{id}, \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp\rangle(\mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ \pi) \\
\mathcal{E}^\sharp[\![\underline{\texttt{rec}}\ e]\!]\ \phi^\sharp\ \pi & = & (\mathsf{lfp}_{\tau_0}(\boldsymbol{\lambda}\tau.\boldsymbol{\lambda}\pi.\langle \mathsf{id}, \tau\rangle(\mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp\ \pi)))(\pi) \\
 & & \textsf{where}\ \tau_0 = (\boldsymbol{\lambda}\rho.(\mathsf{up}\perp, \dots, \mathsf{up}\perp))^\sharp
\end{array}
$$

$\mathcal{P}^\sharp[\![f_1 = \lambda e_1; \dots; f_n = \lambda e_n]\!]$ yields a non-standard environment containing the least abstractions of the functions in $\mathcal{P}[\![f_1 = \lambda e_1; \dots; f_n = \lambda e_n]\!]$.

$$
\begin{array}{rcl}
\mathcal{P}^\sharp[\![\mathbf{dbp}]\!] & : & \mathbf{FunEnv}^\sharp \\
\multicolumn{3}{l}{\mathcal{P}^\sharp[\![f_1 = \lambda e_1; \dots; f_n = \lambda e_n]\!]} \\
 & = & \mathsf{lfp}(\boldsymbol{\lambda}\phi^\sharp.\varnothing[f_1 \mapsto \boldsymbol{\lambda}\pi.\mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp\ \pi \uparrow 2] \cdots \\
 & & \qquad\qquad [f_n \mapsto \boldsymbol{\lambda}\pi.\mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp\ \pi \uparrow 2])
\end{array}
$$

Table 5.3: The semantic functions $\mathcal{E}^\sharp$ and $\mathcal{P}^\sharp$.

5.20 Definition *Let ϕ be a function environment and let $\phi^\sharp$ be a non-standard function environment, then $\phi^\sharp$ is* correct with respect to ϕ *iff $\phi(f)^\sharp = \phi^\sharp(f)$, for all $f \in$* **fun**.

5.21 Correctness of the non-standard semantics

1. *If $\phi^\sharp$ is correct wrt ϕ, then $(\mathcal{E}[\![e_0]\!]\ \phi)^\sharp = \mathcal{E}^\sharp[\![e_0]\!]\ \phi^\sharp$, for all $e_0 \in$* **dbe***,*

2. *$\mathcal{P}^\sharp[\![f_1\ p_1 = e_1; \dots; f_n\ p_n = e_n]\!]$ is correct wrt $\mathcal{P}[\![f_1\ p_1 = e_1; \dots; f_n\ p_n = e_n]\!]$.*

Proof Ad 1) The proposition is proved by structural induction on e_0. We confine ourselves to a few cases since the main prerequisites have already been developed in the preceding sections. Case $e_0 = k.i$:

$$
\begin{array}{rcll}
(\mathcal{E}[\![k.i]\!]\ \phi)^\sharp & = & (\mathsf{on}_i \circ \mathsf{acc}_k)^\sharp & \text{def. } \mathcal{E} \\
 & = & \mathsf{acc}_k^\sharp \circ \mathsf{on}_i^\sharp & \text{Lemma 5.8} \\
 & = & \langle k\colon \cdot\rangle \circ (i\colon \cdot) & \text{Lemma 5.13} \\
 & = & \mathcal{E}^\sharp[\![k.i]\!]\ \phi^\sharp & \text{def. } \mathcal{E}^\sharp
\end{array}
$$

Case $e_0 = \texttt{freeze}\ e$ or $e_0 = \texttt{unfreeze}\ e$: Analogous using Lemma 5.11 and the induction hypothesis. Case $e_0 = s\ e$ or $e_0 = c\ e$: Analogous. Case $e_0 = f\ e$: Analogous using $\phi(f)^\sharp = \phi^\sharp(f)$. Case $e_0 = e_1 \ldots e_n$:

$$\begin{array}{rcll} (\mathcal{E}[\![e_1 \ldots e_n]\!]\ \phi)^\sharp & = & \langle \mathcal{E}[\![e_1]\!]\ \phi, \ldots, \mathcal{E}[\![e_n]\!]\ \phi \rangle^\sharp & \text{def. } \mathcal{E} \\ & = & \langle (\mathcal{E}[\![e_1]\!]\ \phi)^\sharp, \ldots, (\mathcal{E}[\![e_n]\!]\ \phi)^\sharp \rangle & \text{Lemma 5.13} \\ & = & \langle \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp, \ldots, \mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp \rangle & \text{ind. hyp.} \\ & = & \mathcal{E}^\sharp[\![e_1 \ldots e_n]\!]\ \phi^\sharp & \text{def. } \mathcal{E}^\sharp \end{array}$$

Case $e_0 = \underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1 \mid \cdots \mid c_n \to e_n$: Let $\varphi_i = \mathbf{out}_{c_i} \circ \mathcal{E}[\![e]\!]\ \phi$ and let $\tau_i(\pi) = \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ [c_i \colon \pi]$. We start by proving $\varphi_i^\sharp = \tau_i$.

$$\begin{array}{rcll} \varphi_i^\sharp & = & (\mathbf{out}_{c_i} \circ \mathcal{E}[\![e]\!]\ \phi)^\sharp & \text{def. } \varphi_i \\ & = & (\mathcal{E}[\![e]\!]\ \phi)^\sharp \circ \mathbf{out}_{c_i}^\sharp & \text{Lemma 5.8} \\ & = & \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp \circ [c_i \colon \cdot] & \text{ind. hyp. and Lemma 5.15} \\ & = & \tau_i & \text{def. } \tau_i \end{array}$$

Using the alternative specification of $\underline{\texttt{case}}$-expressions given in Section 5.5 we get,

$$\begin{array}{rll} & (\mathcal{E}[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1 \mid \cdots \mid c_n \to e_n]\!]\ \phi)^\sharp & \\ = & (\mathcal{E}[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1\ ?\ \cdots\ ?\ \underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_n \to e_n]\!]\ \phi)^\sharp & \text{by (5.7)} \\ = & ((\mathcal{E}[\![e_1]\!]\ \phi \circ \langle \mathbf{id}, \varphi_1 \rangle) \boxplus \cdots \boxplus (\mathcal{E}[\![e_n]\!]\ \phi \circ \langle \mathbf{id}, \varphi_n \rangle))^\sharp & \text{def. } \mathcal{E} \\ = & (\mathcal{E}[\![e_1]\!]\ \phi \circ \langle \mathbf{id}, \varphi_1 \rangle)^\sharp \sqcup \cdots \sqcup (\mathcal{E}[\![e_n]\!]\ \phi \circ \langle \mathbf{id}, \varphi_n \rangle)^\sharp & \text{Lemma 5.15} \\ = & \langle \mathbf{id}, \varphi_1 \rangle^\sharp \circ (\mathcal{E}[\![e_1]\!]\ \phi)^\sharp \sqcup \cdots \sqcup \langle \mathbf{id}, \varphi_n \rangle^\sharp \circ (\mathcal{E}[\![e_n]\!]\ \phi)^\sharp & \text{Lemma 5.8} \\ = & \langle \mathbf{id}, \tau_1 \rangle \circ \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp \sqcup \cdots \sqcup \langle \mathbf{id}, \tau_n \rangle \circ \mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp & \text{Lemma 5.13 and ind. hyp.} \\ = & \mathcal{E}^\sharp[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1 \mid \cdots \mid c_n \to e_n]\!]\ \phi^\sharp & \text{def. } \mathcal{E}^\sharp \end{array}$$

Case $e_0 = \underline{\texttt{let}}\ e_1\ \underline{\texttt{in}}\ e$: Follows by a straightforward calculation.

$$\begin{array}{rcll} (\mathcal{E}[\![\underline{\texttt{let}}\ e_1\ \underline{\texttt{in}}\ e]\!]\ \phi)^\sharp & = & (\mathcal{E}[\![e]\!]\ \phi \circ \langle \mathbf{id}, \mathcal{E}[\![e_1]\!]\ \phi \rangle)^\sharp & \text{def. } \mathcal{E} \\ & = & \langle \mathbf{id}, \mathcal{E}[\![e_1]\!]\ \phi \rangle^\sharp \circ (\mathcal{E}[\![e]\!]\ \phi)^\sharp & \text{Lemma 5.8} \\ & = & \langle \mathbf{id}, \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp \rangle \circ \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp & \text{Lemma 5.13 and ind. hyp.} \\ & = & \mathcal{E}^\sharp[\![\underline{\texttt{let}}\ e_1\ \underline{\texttt{in}}\ e]\!]\ \phi^\sharp & \text{def. } \mathcal{E}^\sharp \end{array}$$

Case $e_0 = \underline{\texttt{rec}}\ e$: First note that

$$\boldsymbol{\lambda}\rho.\, \mathbf{lfp}_{v_0}(\boldsymbol{\lambda}v.\mathcal{E}[\![e_1]\!]\ \phi\ \langle \rho, v \rangle) \quad = \quad \mathbf{lfp}_{\varphi_0}(\boldsymbol{\lambda}\varphi.\mathcal{E}[\![e_1]\!]\ \phi \circ \langle \mathbf{id}, \varphi \rangle)$$

with $\varphi_0 = \boldsymbol{\lambda}\rho.v_0$. Since the least abstraction of the functional $\boldsymbol{\lambda}\varphi.\mathcal{E}[\![e_1]\!]\ \phi \circ \langle \mathbf{id}, \varphi \rangle$ is, in fact, $\boldsymbol{\lambda}\tau.\langle \mathbf{id}, \tau \rangle \circ \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp$ the equation follows by Lemma 5.19.

Ad 2) First note that $\bigsqcup\{\, \pi_2 \mid \pi_1 \otimes \pi_2 \sqsubseteq \pi \,\} = \pi \uparrow 2$. The proposition follows using a generalization of Lemma 5.19 to multiple fixpoints. ■

5.8 Bibliographic notes

The only work we know of which considers the question of best or least abstractions is that of Davis [46, 47]. He shows in *loc. cit.* that stable functions possess least abstractions and that every stable, strict, and bottom-reflecting[3] function is determined by its least abstraction.[4] Theorem 5.3 improves on his result—at least in a first-order setting—by replacing the condition of stability with the requirement that the domain of the function must satisfy Axiom I. However, the latter condition is not very restrictive since it is satisfied by every domain obtained as the solution of a first-order domain equation.

Note that the additional restriction of Davis to bottom-reflecting functions is necessary since he works with the function space $D_\perp \circ\!\!\rightarrow E_\perp$ instead of $D_\perp \circ\!\!\rightarrow E$ as we do. This approach is omnipresent in the literature on projection-based analysis being probably motivated by the categorical view of $(\cdot)_\perp$ as a functor. We feel, however, that it is much more natural both from a theoretical and a practical point of view to work with $D_\perp \circ\!\!\rightarrow E$ given by the domain isomorphism $D \rightarrow E \cong D_\perp \circ\!\!\rightarrow E$.

We are not aware of any work that deals with the problem of determining the least abstraction of a textual function. The non-standard semantics specified, for example, in [105] or [150] yields generally only a safe approximation of the least abstraction. The imprecision is, of course, caused by the use of independent projections. Davis and Wadler [50] also note the inaccuracy of [150]. Albeit they propose quite a different solution. Although not stated explicitly the non-standard semantics of a function, say, $\varphi : D \rightarrow E$ is given in *loc. cit.* by a projection-transformer transformer of type $\forall\delta.(\|D\| \rightarrow \|\delta\|) \rightarrow (\|E\| \rightarrow \|\delta\|)$. Hence, their backward analysis is really a forward analysis at a higher level of abstraction! The two concepts of abstractions are, however, not unrelated. Let $\varphi^\sharp$ be the abstraction of φ in our sense and let φ^f be the abstraction in the sense of Davis and Wadler, then we have $\varphi^f(\tau_1, \ldots, \tau_n) \sqsupseteq \langle\tau_1, \ldots, \tau_n\rangle \circ \varphi^\sharp$. The inclusion is generally proper since the analysis of Davis and Wadler is high fidelity but not relational.

[3] A function, say, φ is bottom-reflecting iff $(\forall x)\ \varphi(x) = \perp \Longrightarrow x = \perp$.

[4] The observation that a stable function possesses a least abstraction is attributed to Sebastian Hunt.

Chapter 6

Contexts

Nous voudrions que la Nature n'allât pas plus loin, qu'elle fût finie, comme notre esprit: mais ce n'est point connaître la grandeur et la majesté de l'Auteur des choses.

— Gottfried Wilhelm Leibniz, *Letter to S. Clarke (1715)*

This and the next chapter develop an effective variant of the non-standard semantics (termed approximating semantic) which may be used as an integral part of a compiler for non-strict functional languages. Since strictness is a non-trivial property of programs the approximating semantics generally does not detect all cases of strictness. Albeit its results are sound in the sense that they never falsely signal strictness. Hence it is safe to employ them in an optimizing compiler. The quality of the optimizer is naturally influenced by the accuracy of the inferred strictness properties. It is thus worthwhile to identify and assess the sources of imprecision. This undertaking is not as difficult as one might expect since we already know the optimal rules!

The first design decision concerns the representation of projections, termed contexts for lack of names, which will be discussed at length in this chapter. We build, of course, upon the results of Chapter 4. Projections on lifts and coalesced sums pose little problems. Lemma 4.8 shows that strict and lazy lift suffice to represent projections on the lift $D_\perp$ provided one knows how to represent projections on the underlying domain D. A similar result holds for coalesced sums where each projection may be defined in terms of $\cdot \oplus \cdots \oplus \cdot$.

Turning to smash products there is a choice to be made. We have seen in Section 4.4 that each projection on a smash product may be represented by a set of independent projections or more precisely by a set of atomic projections. Section 4.5 tells us that independent projections are enough to express strictness properties such as lift strictness in the i-th argument while they fail, for instance, to characterize joint strictness of several arguments. A strictness test employing sets of independent projections amounts to a relational analysis while the restriction to independent projections accordingly corresponds to an independent attribute analysis.

Let us briefly investigate the implications of the two possibilities. It is immediate that the former choice greatly enlarges the size of the abstract domains. Consider, for example, the smash product $D_1 \otimes \cdots \otimes D_n$ with $D_i = \mathbf{2}_\perp$. We have two atomic projections on $\mathbf{2}_\perp$, whence $\mathbf{2}^n$ atoms in $\|D_1 \otimes \cdots \otimes D_n\|$ and 2^{2^n} sets of atoms. Thus the size of the abstract domain grows double exponentially to the length of the sequence. For $n = 5$ there are 4.294.987.296 sets of atoms representing 2.844.972 different projections. On the other side the number of independent projections amounts to $3^n + 1$. For $n = 5$ there are 243 independent projections.

With regard to the generation of code it is interesting to note that the high fidelity analysis provides no additional information which could be utilized by the code generator. The following definitions may serve as an example.

```
if x y z = case x of True → y | False → z
flip x y = if x y (-y)
```

Recall that `if` is strict in its first argument and joint strict in the second and third argument. Albeit the latter information is of no use for the optimizer since neither the second nor the third parameter may be safely passed by value. Hence as far as the code generation is concerned the information provided by the low fidelity analysis is sufficient. This does, of course, not imply that the high fidelity analysis is virtually useless as the example of `flip` shows. The function `flip` is obviously strict in both arguments but an analysis based on independent projections only discovers strictness in the first argument since the analysis of `flip` uses the low fidelity context transformer of `if` which inaccurately classifies `y` as probably not needed.

In view of these facts and since we would rather trade accuracy for efficiency than the other way round we confine ourselves to independent projections. However, it should be stressed that the approximating semantics could be easily extended to a relational analysis.

It remains to discuss the representation of projections on

1. system-defined types,
2. polymorphic types and
3. recursive types.

Let us consider each point in turn. Recall that system-defined types like ε, `num` or `char` are interpreted by flat domains. Since we do not want the analysis to depend on concrete values, primitive types are treated like the two-point domain **2**. Hence, we differentiate merely between proper and improper values. This is completely analogous to the situation in forward analysis. Hence bottom and identity are the only projections respective contexts on system-defined types. Though this approximation is rather crude it plays little rôle in practice since the additional information gained by more complex abstract domains again cannot be put to use by the code generator.

The choice of contexts for polymorphic types is best motivated by means of an example.

```
dup :: [num] → ([num],[num])
dup a = (a,a)
```

The function `dup` yields a pair containing the function's argument in both components. Note that we have restricted the most general type of `dup`, $\forall\alpha.\alpha \to (\alpha, \alpha)$, to a monomorphic instance. If we confine ourselves to independent projections the least abstraction of `dup` satisfies

$$\begin{aligned}
\mathtt{dup}^\sharp(\pi_{\oplus} \otimes \pi'_{\oplus}) &= (\pi \mathbin{\&} \pi')_{\oplus} \\
\mathtt{dup}^\sharp(\pi_{\oplus} \otimes \pi'_{\bot}) &= (\pi \sqcup \pi \mathbin{\&} \pi')_{\oplus} \\
\mathtt{dup}^\sharp(\pi_{\bot} \otimes \pi'_{\oplus}) &= (\pi \mathbin{\&} \pi' \sqcup \pi')_{\oplus} \\
\mathtt{dup}^\sharp(\pi_{\bot} \otimes \pi'_{\bot}) &= (\pi \sqcup \pi')_{\bot}.
\end{aligned}$$

Having concrete representations of projections on `[num]` and an operation which realizes the conjunction of contexts the context transformer for `dup` is finite and easily tabulated. However, if we drop the type restriction and analyse `dup` at its most general type we are faced with the problem that the inferred context transformer must encompass all instances of $\forall\alpha.\alpha \to (\alpha, \alpha)$. We may, of course, analyse dup repeatedly, for every monomorphic instance, which occurs in the program, but this is clearly a very inefficient solution.

The polymorphism which we consider is sometimes classified as *parametric polymorphism* since the quantified type variable may be regarded as an implicit type parameter to the respective function. This suggests to parameterize the context transformer of a polymorphic function with context variables and to interpret $\&$ and $\sqcup$ symbolically. Hence, a context on a generic type variable is a formal expression involving context variables and the symbols & and + (we decided to distinguish between the syntactic symbols & and + and their semantic counterparts $\&$ and $\sqcup$). The context transformer for dup is consequently defined by

$$\begin{array}{lcl}
\mathtt{dup}^{\alpha}(\gamma_1!, \gamma_2!) & = & (\gamma_1 \,\texttt{\&}\, \gamma_2)! \\
\mathtt{dup}^{\alpha}(\gamma_1!, \gamma_2?) & = & (\gamma_1 \,\texttt{+}\, \gamma_1 \,\texttt{\&}\, \gamma_2)! \\
\mathtt{dup}^{\alpha}(\gamma_1?, \gamma_2!) & = & (\gamma_1 \,\texttt{\&}\, \gamma_2 \,\texttt{+}\, \gamma_2)! \\
\mathtt{dup}^{\alpha}(\gamma_1?, \gamma_2?) & = & (\gamma_1 \,\texttt{+}\, \gamma_2)?.
\end{array}$$

We will show in Section 6.1 that formal context expressions of the kind used above form a finite lattice.

A recursive type generally gives rise to an infinite lattice of projections. Consider the domain **list** D of finite and infinite lists defined by the equation **list** $D \cong \mathbf{1}_\perp \oplus D_\perp \otimes (\textbf{list}\ D)_\perp$. The corresponding projection lattice $\|$ **list** $D\|$ contains, of course, the counterparts of H, T, HT, and L defined in the introduction, but also such 'exotic' projections as one which evaluates exactly those elements of its argument list whose index is prime. Since the analysis of a program involves a fixpoint computation we must ensure that the underlying abstract domains satisfy the ascending chain condition which in turn is granted if they are of finite length or even finite.[1] In order to satisfy this need we confine ourselves to *uniform projections* respective contexts. A uniform projection defines the same degree of evaluation on the whole argument as on each of its recursive components excluding projections such as the one described above. A projection π on **list** D, for example, is uniform if and only if it satisfies

$$\pi \quad = \quad \pi_1 \oplus (\pi_2)_{\ell_1} \otimes \pi_{\ell_2},$$

for some $\pi_1 \in \|\mathbf{1}_\perp\|$, $\pi_2 \in \|D\|$ and $\ell_1, \ell_2 \in \{\textcircled{\perp}, \perp\}$. A non-uniform projection, say, π has to be approximated by the least uniform projection above π.

The restriction to uniform contexts excludes many interesting contexts. However, this approach is sufficiently general with regard to the generation of code. Consider the translation of a structural recursive function. If the recursive calls are not situated in the same context as the body of the function we are forced to specialize each of these calls leading to a combinatorial explosion of code variants. Furthermore, many structural recursive functions, for example, `append`, `length` or `sum` behave in fact uniform with respect to the recursive components.

Plan of the chapter Section 6.1 develops the theory of formal context expressions dealing with syntax, semantics, order and normalform. The results presented are generalized in the subsequent sections (6.2 and 6.3) to contexts on arbitrary types. Finally, Section 6.4 shows how to define approximative variants of $\&$ and $\sqcup$.

[1] This is not quite true. Using the technique of widening and narrowing, see [41], one may also use infinite abstract domains.

6.1 Polymorphic contexts

Contexts on generic type variables are formal expressions build from context variables, the constant `bot` and the connectives `&` and `+`. Expressions of this kind are called polymorphic contexts in the sequel. We have motivated in the introduction to this chapter why this is, in fact, an appropriate choice.

As the reader might expect this section has a rather algebraic flavour especially since a part of the algebraic routine program (syntax, semantics, normalform, word problem etc) is touched.

6.1.1 Syntax and semantics

The syntactic categories of polymorphic contexts are shown in Table 6.1.

meta variable		category	comment
γ	$\in$	**cvar**	context variables
κ	$\in$	**cnt**	polymorphic contexts

Table 6.1: Syntactic categories of polymorphic contexts

6.1 Definition *Let $C \subseteq$ **cvar** be a set of context variables. Then κ is a* polymorphic context with variables in C *iff $C \vdash \kappa : \mathbf{cnt}$ is derivable using the deduction rules below.*

$$\frac{}{C \vdash \mathtt{bot} : \mathbf{cnt}} \qquad \frac{}{C \cup \{\gamma\} \vdash \gamma : \mathbf{cnt}}$$

$$\frac{C \vdash \kappa : \mathbf{cnt} \quad C \vdash \kappa' : \mathbf{cnt}}{C \vdash \kappa \mathbin{\&} \kappa' : \mathbf{cnt}} \qquad \frac{C \vdash \kappa : \mathbf{cnt} \quad C \vdash \kappa' : \mathbf{cnt}}{C \vdash \kappa + \kappa' : \mathbf{cnt}}$$

The set of all polymorphic contexts wrt C is denoted by $\mathbf{cnt}(C)$.

In order to save some parenthesis it is understood that `&` has a higher precedence than `+`. Hence, $\gamma_1 \mathbin{\&} \gamma_2 + \gamma_3$ is an abbreviation for $(\gamma_1 \mathbin{\&} \gamma_2) + \gamma_3$.

Polymorphic contexts denote projections on an arbitrary domain. The interpretation of context variables is given by an *assignment* which maps variables to projections. The symbols `bot`, `&` and `+` are interpreted in the obvious way. Table 6.2 summarizes the semantic equations for polymorphic contexts.

Syntactically, the forms $\gamma_1 + \gamma_2$ and $\gamma_2 + (\gamma_1 \mathbin{\&} \gamma_2 + \gamma_1)$ are considered as different polymorphic contexts, even though they are equivalent in the sense that they always denote the same projection irrespective of the assignment. This motivates the following definition.

6.2 Definition *Let $C \subseteq$ **cvar** and let $\kappa, \kappa' \in \mathbf{cnt}(C)$ be polymorphic contexts. Then*

$$\begin{aligned} \kappa \approx \kappa' \quad &:\Longleftrightarrow \quad (\forall \nu)\ \mathcal{C}[\![\kappa]\!]\nu = \mathcal{C}[\![\kappa']\!]\nu, \\ \kappa \lessapprox \kappa' \quad &:\Longleftrightarrow \quad (\forall \nu)\ \mathcal{C}[\![\kappa]\!]\nu \sqsubseteq \mathcal{C}[\![\kappa']\!]\nu. \end{aligned}$$

It is easy to see that $\lessapprox$ defines a pre-order on $\mathbf{cnt}(C)$. Since $\approx$ is the equivalence relation induced by $\lessapprox$, that is, $\kappa \approx \kappa' \Longleftrightarrow \kappa \lessapprox \kappa' \wedge \kappa' \lessapprox \kappa$ we have the following

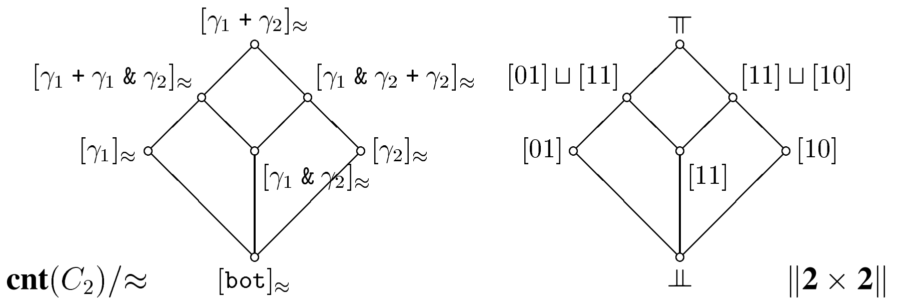
Let ν be an assignment mapping context variables to projections. Then $\mathcal{C}[\![\kappa]\!]\nu$ denotes the value of the polymorphic context κ with respect to the assignment ν.

$$\begin{aligned}
\mathcal{C}[\![\texttt{bot}]\!]\nu &= \perp\!\!\!\perp \\
\mathcal{C}[\![\gamma]\!]\nu &= \nu(\gamma) \\
\mathcal{C}[\![\kappa \mathbin{\&} \kappa']\!]\nu &= \mathcal{C}[\![\kappa]\!]\nu \mathbin{\&} \mathcal{C}[\![\kappa']\!]\nu \\
\mathcal{C}[\![\kappa + \kappa']\!]\nu &= \mathcal{C}[\![\kappa]\!]\nu \sqcup \mathcal{C}[\![\kappa']\!]\nu
\end{aligned}$$

Table 6.2: The semantics of polymorphic contexts

6.3 Fact *Let* $C \subseteq \mathbf{cvar}$, *then* $\langle \mathbf{cnt}(C)/\approx; \lessapprox\rangle$ *is a partial order.*

Let $C_n := \{\gamma_1, \ldots, \gamma_n\}$ be a set of n distinct context variables. The partial order $\mathbf{cnt}(C_2)/\approx$ is displayed below.



If we interpret the elements of $\mathbf{cnt}(C_2)$ by projections on $\mathbf{2} \times \mathbf{2}$ such that $\nu(\gamma_1) = [01]$ and $\nu(\gamma_2) = [10]$, then non-equivalent contexts are interpreted, in fact, by different projections. The lattice $\|\mathbf{2} \times \mathbf{2}\|$ displayed on the right is even isomorphic to $\mathbf{cnt}(C_2)/\approx$. We will show in the next section that the correspondence between the equivalence classes $[\cdot]_\approx$ and projections on $\mathbf{2}^n$ holds in general which implies that $\mathbf{cnt}(C_n)/\approx$ forms a finite lattice.

6.1.2 Order and normalform

The semantic relations $\approx$ und $\lessapprox$ are of central importance to the implementation of the approximating semantics. In order to realize the fixpoint computation we have to know when two elements of the abstract domain are equivalent. This amounts to the *word problem* for 'projection algebras'. Furthermore, we must ensure that the operations involved in the fixpoint computation are monotonic wrt $\lessapprox$ in order to guarantee the existence of a least fixpoint.

This section gives a syntactic or if you like proof-theoretic characterization of both relations. Thereby we proceed in two steps. We start with a formal calculus based on the laws listed in Lemma 4.23. Applying the laws in a systematic way yields a normalform for polymorphic contexts akin to lattice or Boolean polynomials. The normalform allows to construct a projection lattice in which non-equivalent contexts denote different projections (generalizing the example at the end of the last section). This result greatly simplifies the proof of the Adequacy Theorem which establishes the equivalence of the semantic and syntactic notions of order and equality.

6.4 Definition *The relations $\lesssim$ and $\sim$ on polymorphic contexts are defined by the following axioms plus the usual rules for equality (reflexivity, symmetry, transitivity and congruence).*

$$
\begin{array}{rcl}
\mathtt{bot} + \kappa & \sim & \kappa \\
\mathtt{bot} \,\&\, \kappa & \sim & \mathtt{bot} \\
(\kappa_1 + \kappa_2) + \kappa_3 & \sim & \kappa_1 + (\kappa_2 + \kappa_3) \\
(\kappa_1 \,\&\, \kappa_2) \,\&\, \kappa_3 & \sim & \kappa_1 \,\&\, (\kappa_2 \,\&\, \kappa_3) \\
\kappa_1 + \kappa_2 & \sim & \kappa_2 + \kappa_1 \\
\kappa_1 \,\&\, \kappa_2 & \sim & \kappa_2 \,\&\, \kappa_1 \\
\kappa + \kappa & \sim & \kappa \\
\kappa \,\&\, \kappa & \sim & \kappa \\
\kappa_1 \,\&\, (\kappa_2 + \kappa_3) & \sim & \kappa_1 \,\&\, \kappa_2 + \kappa_1 \,\&\, \kappa_3 \\
(\kappa_1 + \kappa_2) \,\&\, \kappa_3 & \sim & \kappa_1 \,\&\, \kappa_3 + \kappa_2 \,\&\, \kappa_3 \\
\kappa_1 + \kappa_2 & \sim & \kappa_1 + \kappa_1 \,\&\, \kappa_2 + \kappa_2
\end{array}
\qquad
\frac{\kappa + \kappa' \sim \kappa'}{\kappa \lesssim \kappa'}
$$

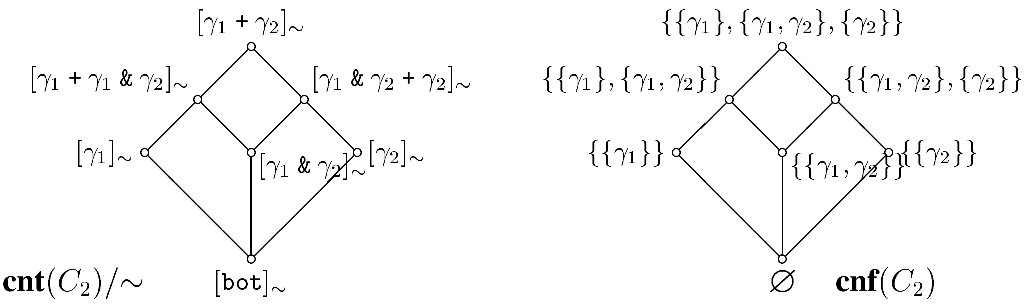
Note that $\sim$ defines an equivalence relation on $\mathbf{cnt}(C)$. It is in fact the equivalence relation induced by $\lesssim$, that is, $\kappa \sim \kappa' \iff \kappa \lesssim \kappa' \wedge \kappa' \lesssim \kappa$.

6.5 Lemma *Let $C \subseteq \mathbf{cvar}$, then $\langle \mathbf{cnt}(C)/{\sim};\ \lesssim\rangle$ forms a pointed $\sqcup$-semilattice where $[\mathtt{bot}]_\sim$ is the least element and $[\kappa + \kappa']_\sim = [\kappa]_\sim \sqcup [\kappa']_\sim$.*

Proof Let $[\kappa]_\sim + [\kappa']_\sim := [\kappa + \kappa']_\sim$ and $[\kappa]_\sim \preccurlyeq [\kappa']_\sim :\iff [\kappa]_\sim + [\kappa']_\sim = [\kappa']_\sim$. It is easy to see that $+$ is idempotent, associative, commutative and has the neutral element $[\mathtt{bot}]_\sim$. Hence $\langle \mathbf{cnt}(C)/{\sim};\ \preccurlyeq\rangle$ forms a pointed $\sqcup$-semilattice. It remains to show that $\lesssim$ and $\preccurlyeq$ define the same relation.

$$
\begin{array}{rcll}
\kappa \lesssim \kappa' & \iff & \kappa + \kappa' \sim \kappa' & \text{def. } \lesssim \\
& \iff & [\kappa + \kappa']_\sim = [\kappa']_\sim & \\
& \iff & [\kappa]_\sim \preccurlyeq [\kappa']_\sim & \text{def. } + \text{ and } \preccurlyeq \ \blacksquare
\end{array}
$$

Using the fact that & distributes over + polymorphic contexts may be transformed to disjunctions of conjunctions of type variables. Furthermore the ACI properties of & and + show that conjunctive terms correspond to sets of context variables and disjunctive terms to sets of disjunctions respectively. There are 3 conjuncts in $\mathbf{cnt}(C_2)/{\sim}$, namely γ_1, γ_2 and $\gamma_1 \,\&\, \gamma_2$, and since $\gamma_1 + \gamma_2 \sim \gamma_1 + \gamma_1 \,\&\, \gamma_2 + \gamma_2$ a total of $2^3 - 1 = 7$ contexts. The Hasse diagram of $\mathbf{cnt}(C_2)$ is displayed below.



The diagram on the right shows the set representation of contexts which we will adopt in the sequel. Let $\wp^\circ_{\mathrm{f}}(X) := \wp_{\mathrm{f}}(X) \setminus \{\varnothing\}$. The set

$$\{\{\gamma_{11}, \ldots, \gamma_{1n_1}\}, \ldots, \{\gamma_{m1}, \ldots, \gamma_{mn_m}\}\} \subseteq \wp_{\mathrm{f}}(\wp^\circ_{\mathrm{f}}(C))$$

represents the context

$$\gamma_{11} \mathbin{\&} \ldots \mathbin{\&} \gamma_{1n_1} + \ldots + \gamma_{m1} \mathbin{\&} \ldots \mathbin{\&} \gamma_{mn_m}.$$

The bottom context is represented by the empty set. Note that contrary to disjunctions conjunctions may not be empty, because the set of polymorphic contexts contains no element denoting the conjunctive unit.

Applying the axiom $\kappa_1 + \kappa_2 \sim \kappa_1 + \kappa_1 \mathbin{\&} \kappa_2 + \kappa_2$ to a context κ corresponds to closing the set representation of κ under set union. Sets which are closed under union uniquely determine the equivalence classes $[\cdot]_\sim$ and vice versa. The closure operator which lies at the heart of this argument is in fact a slight variation of the closure operator **c** introduced in Section 4.1 yielding as a by-product the isomorphism of $\mathbf{cnt}(C_n)/\sim$ to $\|\mathbf{2}^n\|$.

6.6 Definition and Lemma *Let $C \subseteq$ **cvar**, the function **n** : $\wp_\mathrm{f}(\wp^\circ_\mathrm{f}(C)) \to \wp_\mathrm{f}(\wp^\circ_\mathrm{f}(C))$ with*

$$\mathbf{n}(\mathcal{A}) = \{ \bigcup \mathcal{B} \mid \varnothing \neq \mathcal{B} \subseteq \mathcal{A} \}$$

is a closure operator. An element of $\mathbf{n}(\wp_\mathrm{f}(\wp^\circ_\mathrm{f}(C)))$ is termed normal polymorphic context. *The set of all normal polymorphic contexts wrt C is denoted $\mathbf{cnf}(C)$. Let $C_n := \{\gamma_1, \ldots, \gamma_n\}$ be a set of n distinct context variables, then $\langle \mathbf{cnf}(C_n);\ \subseteq \rangle$ is isomorphic to $\langle \|\mathbf{2}^n\|;\ \sqsubseteq \rangle$.*

Proof Identify $\mathbf{2}^n$ with $\wp(C_n)$. We show that $\mathbf{cnf}(C_n)$ is isomorphic to the set of all normal substructures $\mathfrak{N}(\wp(C_n))$. By Lemma 4.4 a set M is normal iff $\mathbf{c}(M) = M$. It is not hard to see that $\mathbf{n}(\mathcal{A}) = \mathcal{A}$ if and only if $\mathbf{c}(\mathcal{A} \cup \{\varnothing\}) = \mathcal{A} \cup \{\varnothing\}$. Hence the map $A \mapsto A \cup \{\varnothing\}$ is an isomorphism of $\mathbf{cnf}(C_n)$ onto $\mathfrak{N}(\wp(C_n)) \cong \|\mathbf{2}^n\|$. ■

The acronym **cnf** should not be confused with conjunctive normalform, ist rather stands for context normalform.

As the snappy proof of Lemma 6.6 is based on the formal correspondence between **c** and **n** it probably gives not much insight into the result itself. The lemma is best explained with the use of an auxiliary notation. Let $\mathbf{i} \in \mathbf{2}^n$ with

$$\mathbf{i} = a_1 \ldots a_n \text{ where } a_j = \begin{cases} 1 & \textbf{if } i = j \\ 0 & \textbf{otherwise} \end{cases}$$

denote the binary representation of 2^i. Note that $\mathbf{i}$ for $1 \leqslant i \leqslant n$ is an atom of the Boolean lattice $\mathbf{2}^n$.

The lemma essentially shows that $\|\mathbf{2}^n\|$ serves as the standard model of $\mathbf{cnt}(C_n)/\sim$ since it has the remarkable property that two contexts are provably equal if and only if the denote the same projection wrt the assignment $\nu(\gamma_i) = [\,\mathbf{i}\,]$. The assignment plays, of course, the key rôle in this statement. To see why this particular choice works recall first that a projection lattice is generated by its atoms, that is, $\pi = \bigsqcup\{ [a] \mid a \in F^\circ(\mathbf{im}\,\pi) \}$, for every $\pi \in \|\mathbf{2}^n\|$. Second, since the underlying domain, $\mathbf{2}^n$, is a Boolean lattice we also may represent each element of $\mathbf{2}^n$ by a join of atoms: $a = \bigsqcup\{ \mathbf{i} \in \mathcal{A}(\mathbf{2}^n) \mid \mathbf{i} \sqsubseteq a \}$. The link between projections on $\mathbf{2}^n$ and polymorphic contexts in normalform is finally established by the following law

$$[a] \mathbin{\&} [b] = [a \sqcup b], \tag{6.1}$$

which shows that the conjunction of atomic projections again yields an atomic projection whose non-trivial fixpoint equals the join of the fixpoints of the respective projections. This implies that each atomic projection[2] may be represented by a conjunctions of projections of the form $[\,\mathbf{i}\,]$ with $\mathbf{i} \in \mathcal{A}(\mathbf{2}^n)$. Hence, each projection equals a disjunction of conjunctions of projections of this kind.

The three conjuncts in $\mathbf{cnt}(C_2)/{\sim}$, for example denote the atomic projections $[01]$, $[10]$ and $[01] \mathbin{\&} [10] = [01 \sqcup 10] = [11]$. We have depicted the lattice $\mathbf{cnt}(C_2)/{\sim}$ several times (see above or cf to the end of Section 6.1.1). The lattice $\mathbf{cnt}(C_3)/{\sim}$ contains seven conjuncts whose images in $\mathbf{2}^3$ are displayed in Figure 6.1. [Note that they are ordered according to their non-trivial fixpoint.] The complete lattice $\mathbf{cnt}(C_3)/{\sim}$ contains 61 elements and is already to big to be drawn properly.

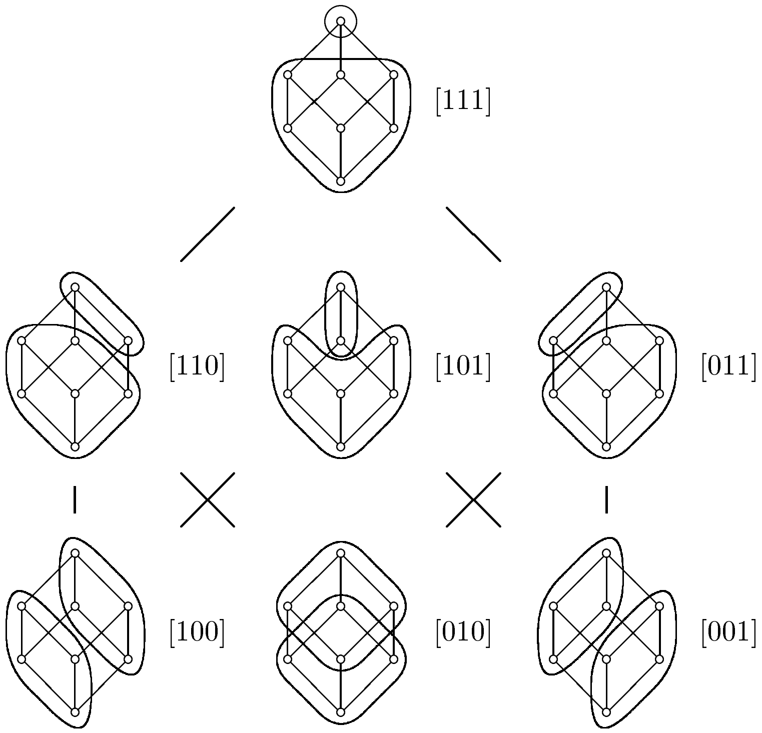


Figure 6.1: Atomic projections on $\mathbf{2} \times \mathbf{2} \times \mathbf{2}$

Equation 6.1 may be explained as follows. Each atomic projection, say, $[a]$ partitions its domain into two blocks, namely the inverse image of $\perp$ and the inverse image of a, which equals $\uparrow a$. We have shown in Section 4.6 that the conjunction of projections forms the union of the former blocks and the intersection of the latter ones. The equation is then implied by $\uparrow a \cap \uparrow b = \uparrow(a \sqcup b)$.

6.7 Lemma *Let D be a domain and let $\varnothing \neq F \Subset F^\circ(D)$. If $\bigsqcup F$ exists in D, then*

$$\&[F] \;=\; [\textstyle\bigsqcup F].$$

Proof We show $[a] \mathbin{\&} [b] = [a \sqcup b]$ for $a, b \in F^\circ(D)$. The proposition follows by a straightforward induction. First note that $a \sqcup b \in F^\circ(D)$ provided $a \sqcup b$ exists. Case $a \sqcup b \sqsubseteq v$: It follows that $[a](v) = a \neq \perp \neq b = [b](v)$. Hence,

$$([a] \mathbin{\&} [b])(v) = ([a] \sqcup [b])(v) = a \sqcup b = [a \sqcup b](v).$$

[2] As in chemistry atoms do not really deserve their name.

Case $a \sqcup b \not\sqsubseteq v$: Consequently $a \not\sqsubseteq v$ or $b \not\sqsubseteq v$ implying $([a] \mathbin{\&} [b])(v) = \bot = [a \sqcup b](v)$. ■

In the sequel and of course in the actual implementation we build upon the normalform of polymorphic contexts. Hence it is useful to define variants of $\&$ and $\sqcup$ which work directly on normalforms. As a by-product we obtain a simple definition of the normalization process. To discriminate between the operators used in the non-standard semantics and their abstract counterparts we annotate the latter ones with the greek letter α which stands for approximation. Hence, $\&$ and $\sqcup$ denote operations on projections, while $\&^\alpha$ and $\sqcup^\alpha$ symbolize their counterparts on contexts. [Not to forget `&` and `+` being syntactic components of polymorphic contexts.]

6.8 Definition *Let $\mathcal{A}, \mathcal{B} \in$ **cnf**(C) be normal polymorphic contexts. The conjunction and disjunction of normal polymorphic contexts is defined by*

$$\begin{aligned} \mathcal{A} \mathbin{\&^\alpha} \mathcal{B} &= \{ A \cup B \mid A \in \mathcal{A} \wedge B \in \mathcal{B} \}, \\ \mathcal{A} \sqcup^\alpha \mathcal{B} &= \mathcal{A} \cup \mathcal{A} \mathbin{\&^\alpha} \mathcal{B} \cup \mathcal{B}. \end{aligned}$$

The function norm : **cnt**$(C) \to$ **cnf**(C) *with*

$$\begin{aligned} \text{norm}[\![\texttt{bot}]\!] &= \varnothing \\ \text{norm}[\![\gamma]\!] &= \{\{\gamma\}\} \\ \text{norm}[\![\kappa \mathbin{\texttt{\&}} \kappa']\!] &= \text{norm}[\![\kappa]\!] \mathbin{\&^\alpha} \text{norm}[\![\kappa']\!] \\ \text{norm}[\![\kappa \mathbin{\texttt{+}} \kappa']\!] &= \text{norm}[\![\kappa]\!] \sqcup^\alpha \text{norm}[\![\kappa']\!] \end{aligned}$$

maps its argument to the context normalform.

The Adequacy Theorem summarizes the results of this section. The normalform of contexts and the standard model $\|\mathbf{2}^n\|$ are employed to show that the semantic and syntactic notions of equality and order coincide.

6.9 Adequacy Theorem for polymorphic contexts *Let $\kappa, \kappa' \in$ **cnt**(C_n) be polymorphic contexts. For all assignments $\bar{\nu}$ with $\bar{\nu}(\gamma_i) = [\,\mathbf{i}\,]$ for $1 \leqslant i \leqslant n$ the following holds,*

$$\begin{aligned} \kappa \approx \kappa' &\iff \mathcal{C}[\![\kappa]\!]\bar{\nu} = \mathcal{C}[\![\kappa']\!]\bar{\nu} \iff \kappa \sim \kappa' \iff \text{norm}[\![\kappa]\!] = \text{norm}[\![\kappa']\!], \\ \kappa \lessapprox \kappa' &\iff \mathcal{C}[\![\kappa]\!]\bar{\nu} \sqsubseteq \mathcal{C}[\![\kappa']\!]\bar{\nu} \iff \kappa \lesssim \kappa' \iff \text{norm}[\![\kappa]\!] \subseteq \text{norm}[\![\kappa']\!]. \end{aligned}$$

*Hence, $\langle$**cnt**$(C_n)/{\approx};\ \lessapprox\rangle$, $\langle\|\mathbf{2}^n\|;\ \sqsubseteq\rangle$, $\langle$**cnt**$(C_n)/{\sim};\ \lesssim\rangle$ and $\langle$**cnf**$(C_n);\ \subseteq\rangle$ are isomorphic finite lattices.*

Proof We show the two propositions simultaneously. $(1) \Longrightarrow (2)$: Immediate since $\bar{\nu}$ is an assignment. $(3) \Longrightarrow (1)$: The correctness of the axioms follows by Lemma 4.23 and some simple calculations. $(4) \Longrightarrow (3)$: Immediate since norm$[\![\kappa]\!] \sim \kappa$. $(2) \Longrightarrow (4)$: Identify $\mathbf{2}^n$ with $\wp(C_n)$, then the assignment $\bar{\nu}$ reads as $\bar{\nu}(\gamma_i) = [\{\gamma_i\}]$.

$$\begin{aligned} \mathcal{C}[\![\kappa]\!]\bar{\nu} &= \mathcal{C}(\text{norm}[\![\kappa]\!])\bar{\nu} && \text{norm}[\![\kappa]\!] \sim \kappa \text{ and } (3) \Longrightarrow (2) \\ &= \bigsqcup\{ \textstyle\mathop{\&}\{ [\{\gamma\}] \mid \gamma \in A \} \mid A \in \text{norm}[\![\kappa]\!] \} && \text{def. } \mathcal{C} \\ &= \bigsqcup\{ [A] \mid A \in \text{norm}[\![\kappa]\!] \} && \text{Lemma 6.7} \\ &= \pi_{\text{norm}[\![\kappa]\!]} && \text{Theorem 4.6} \end{aligned}$$

Hence $\mathcal{C}[\![\kappa]\!]\bar{\nu} = \mathcal{C}[\![\kappa']\!]\bar{\nu}$ implies $\pi_{\text{norm}[\![\kappa]\!]} = \pi_{\text{norm}[\![\kappa']\!]}$ and since $N \mapsto \pi_N$ is an embedding we have norm$[\![\kappa]\!] =$ norm$[\![\kappa']\!]$. Analogous for '$\subseteq$'. ■

We conclude the section with some considerations on the size and the length of polymorphic context lattices. Especially the length of **cnf**(C_n) is of some interest since it attributes to the upper bound for the number of iterations in a fixpoint computation (see Section 7.7). Recall that the length of a partial order P is the number of elements minus one of the longest chain in P. Since **cnf**(C_n) is the closure of the powerset $\wp(\wp^\circ(C_n))$, its length, $2^n - 1$ is an upper bound for the length of **cnf**(C_n). The following lemma shows that the upper bound is in fact reached.

6.10 Lemma *The length of* **cnf**(C_n) *is* $2^n - 1$.

Proof We order the elements of $\wp^\circ(C)$ by decreasing size: Let $(A_i)_{1 \leqslant i \leqslant m}$ with $m = 2^n - 1$ be a sequence such that $(\forall i, j)\ 1 \leqslant i \leqslant j \leqslant m \implies \text{card}(A_i) \geqslant \text{card}(A_j)$. Consequently, the sets $\{ A_i \mid 1 \leqslant i \leqslant k \}$ are closed under union. It immediately follows that the sequence $(\{ A_i \mid 1 \leqslant i \leqslant k \})_{0 \leqslant k \leqslant m}$ of length 2^n is a maximal ascending chain in **cnf**(C). ■

Unfortunately, the size of **cnf**(C_n) is not known in general. Just for curiosity the size for $1 \leqslant n \leqslant 5$ (the known values) is listed below.[3] Note that 2^{2^n-1} is a (very imprecise) upper bound for the number of elements.

n	1	2	3	4	5
card(**cnf**(C_n))	2	7	61	2.480	1.422.486
2^{2^n-1}	2	8	128	32.768	2.147.483.648

To assess Lemma 6.10 note that the number n of context variables which are actually used in a strictness test is determined by the result type of the function to analyse. More precisely n equals the number of occurrences of the *same* type variable in the result type. For example, `append` may be analysed using one context variable, `dup` using two context variables. Since it is quite exceptional that the same variable occurs more than three or four times, the result of Lemma 6.10 has little implications in practice. Furthermore, in one of these rare cases we are still free to reduce complexity (thereby sacrificing accuracy) by identifying two or more context variables.

6.2 Syntax and semantics

On each type we define an abstract domain of contexts which approximates the corresponding (generally infinite) lattice of projections. As for the special case of polymorphic contexts the abstract domains form finite lattices. The syntax of contexts follows very closely the syntax of types. We may, in fact, regard contexts as type expressions incorporating strictness annotations. [This is due to the restriction to independent and uniform projections.]

6.11 Definition [Extension of Definition 6.1] *Let* $\sigma \in$ **texp** *be a type and let* Γ *be a set of assumptions of the form* $\gamma : \|\alpha\|$ *with* $\gamma \in$ **cvar** *and* $\alpha \in$ **tvar**. *We say that* κ *is a context on* σ *wrt* Γ *iff* $\Gamma \vdash \kappa : \|\sigma\|$ *is derivable using the deduction rules below. Let* $\Gamma\restriction_\alpha := \{ \gamma \mid \gamma : \|\alpha\| \in \Gamma \}$.

1. $$\frac{\Gamma\restriction_\alpha \vdash \kappa \in \mathbf{cnt}}{\Gamma \vdash \alpha\kappa : \|\alpha\|} \qquad (\alpha)$$

2. $$\frac{}{\Gamma \vdash \beta : \|\beta\|} \qquad (\beta)$$

[3] The numbers were calculated using a C program due to Hermann Stamm-Wilbrandt.

3. $\overline{\Gamma \vdash \varsigma\texttt{bot} : \|\varsigma\|} \qquad \overline{\Gamma \vdash \varsigma\texttt{ide} : \|\varsigma\|}$ (ς)

4. $\dfrac{\Gamma \vdash \kappa : \|\sigma\|}{\Gamma \vdash \kappa! : \|\sigma?\|} \qquad \dfrac{\Gamma \vdash \kappa : \|\sigma\|}{\Gamma \vdash \kappa? : \|\sigma?\|}$ $(\sigma?)$

5. $\dfrac{\Gamma \vdash \kappa_1 : \|\sigma_1\| \quad \ldots \quad \Gamma \vdash \kappa_n : \|\sigma_n\|}{\Gamma \vdash c_1\kappa_1 \mid \cdots \mid c_n\kappa_n : \|c_1\sigma_1 \mid \cdots \mid c_n\sigma_n\|}$ $(c_1\sigma_1 \mid \cdots \mid c_n\sigma_n)$

6. $\dfrac{\Gamma \vdash \kappa_1 : \|\sigma_1\| \quad \ldots \quad \Gamma \vdash \kappa_n : \|\sigma_n\|}{\Gamma \vdash \kappa_1 \ldots \kappa_n : \|\sigma_1 \ldots \sigma_n\|}$ $(\sigma_1 \ldots \sigma_n$ *with* $n \geqslant 1)$

7. $\dfrac{\Gamma \vdash \kappa : \|\sigma\|}{\Gamma \vdash \mu\beta.\kappa : \|\mu\beta.\sigma\|}$ $(\mu\beta.\sigma)$

The set of all contexts on σ *wrt* Γ *is denoted* $|\Gamma \vdash \sigma|$.

Since the empty sequence type is classified as system-defined there are two contexts on ε, namely $\varepsilon\texttt{bot}$ and $\varepsilon\texttt{ide}$. If we regarded ε as a special case of $\sigma_1 \ldots \sigma_n$ the bottom projection on ε could not be represented.

Let us consider some examples. The identity projection on `bool` is given by False $\varepsilon\texttt{ide}$ | True $\varepsilon\texttt{ide}$. The polymorphic version of the head strict context on lists reads $\mu\beta.\,\texttt{[]}\ \varepsilon\texttt{ide} \mid (\alpha\gamma)?:(\beta)!$. If the type is obvious from the 'context' we will often be sloppy and omit some of the type components, for example, we abbreviate the contexts above by False `ide` | True `ide` and $\mu\beta.\,\texttt{[]}\ \texttt{ide} \mid \gamma?:\beta!$.

Especially generic type variables like α serve primarily as an *aide memoire*. Sometimes we specify them, sometimes we leave them out, whatever is more appropriate. We implicitly drop them, for example, if the corresponding context variables are subject to substitution, for example, $(\mu\beta.\,\texttt{[]}\ \varepsilon\texttt{ide} \mid \alpha\gamma?:\beta!)[\gamma/\texttt{num ide}]$ yields the context $\mu\beta.\,\texttt{[]}\ \varepsilon\texttt{ide} \mid \texttt{num ide}?:\beta!$ with α omitted.

Note that we use μ-bound type variables such as β in $\mu\beta.\,\texttt{[]}\ \texttt{ide} \mid \gamma?:\beta!$ also for context variables. Consequently, an assignment contains bindings for context variables (elements of **cvar**) and for μ-bound type variables (elements of **tcon**). Table 6.3 displays the semantic equations for contexts.

The familiar Substitution Lemma is used several times in the sequel.

6.12 Substitution Lemma *Let* $\beta \in$ **tcon** *and let* $\kappa_1 \in |\Gamma \vdash \sigma_1|$, $\kappa_2 \in |\Gamma \vdash \sigma_2|$, *then*

$$\mathcal{C}(\kappa_1[\beta/\kappa_2])\nu \quad = \quad \mathcal{C}[\![\kappa_1]\!](\nu[\beta \mapsto \mathcal{C}[\![\kappa_2]\!]\nu]).$$

An analogous result holds for $\gamma \in$ **cvar** *and contexts* κ_1 *and* κ_2 *such that* κ_1 *contains no occurrences of* `&` *and* `+`.

Note that the condition on κ_1 is necessary to guarantee that $\kappa_1[\gamma/\kappa_2]$ is a context in the sense of Definition 6.11 since, for instance, $\alpha(\gamma_1\ \texttt{\&}\ \gamma_2)[\gamma_1/\varsigma\texttt{ide}, \gamma_2/\varsigma\texttt{bot}] = \varsigma\texttt{ide}\ \texttt{\&}\ \varsigma\texttt{bot}$ does not result in a proper context.

Definition 6.2 which introduces the relations $\approx$ and $\lessapprox$ generalizes without any modifications to contexts.

Let ν be an assignment mapping context and μ-bound type variables to projections. Then $\mathcal{C}[\![\kappa]\!]\nu$ denotes the value of the context κ with respect to the assignment ν. $$\begin{array}{rcl} \mathcal{C}[\![\alpha\kappa]\!]\nu &=& \mathcal{C}[\![\kappa]\!]\nu \\ \mathcal{C}[\![\beta]\!]\nu &=& \nu(\beta) \\ \mathcal{C}[\![\varsigma\mathtt{bot}]\!]\nu &=& \perp\!\!\!\perp \\ \mathcal{C}[\![\varsigma\mathtt{ide}]\!]\nu &=& \top\!\!\!\top \\ \mathcal{C}[\![\kappa!]\!]\nu &=& (\mathcal{C}[\![\kappa]\!]\nu)_{\oplus} \\ \mathcal{C}[\![\kappa?]\!]\nu &=& (\mathcal{C}[\![\kappa]\!]\nu)_{\perp} \\ \mathcal{C}[\![c_1\kappa_1 \mid \cdots \mid c_n\kappa_n]\!]\nu &=& \mathcal{C}[\![\kappa_1]\!]\nu \oplus \cdots \oplus \mathcal{C}[\![\kappa_n]\!]\nu \\ \mathcal{C}[\![\kappa_1 \ldots \kappa_n]\!]\nu &=& \mathcal{C}[\![\kappa_1]\!]\nu \otimes \cdots \otimes \mathcal{C}[\![\kappa_n]\!]\nu \\ \mathcal{C}[\![\mu\beta.\kappa]\!]\nu &=& \mathbf{lfp}(\boldsymbol{\lambda}\pi.\mathcal{C}[\![\kappa]\!](\nu[\beta \mapsto \pi])) \end{array}$$

Table 6.3: The semantics of contexts (extension of Table 6.2)

6.3 Normalform and order

This section generalizes the results of Section 6.1 to contexts on polymorphic types. Since the projection lattices $\|D_\perp\|$, $\|D_1 \oplus \cdots \oplus D_n\|$, and $\|D_1 \otimes \cdots \otimes D_n\|$ are isomorphic to product lattices, the order relation $\lesssim$ amounts essentially to the coordinatewise order. The only (rather technical) problem is caused by contexts on sequence types. Since the projection $\pi_1 \otimes \cdots \otimes \pi_n$ equals $\perp\!\!\!\perp$ if only one of the components π_i equals $\perp\!\!\!\perp$, we have, for instance, $\mathtt{bot}!\mathtt{ide}? \lessapprox \mathtt{ide}?\mathtt{bot}!$. In other words the operator $\cdot \otimes \cdots \otimes \cdot$ is not free contrary to $(\cdot)_{\oplus}$, $(\cdot)_{\perp}$, and $\cdot \oplus \cdots \oplus \cdot$. As an unwelcome consequence of this fact the pointwise join of contexts is not independent of the equivalence classes' representatives, for example, both $\mathtt{bot}!\mathtt{ide}?$ and $\mathtt{ide}?\mathtt{bot}!$ denote the bottom projection but the coordinatewise join yields

$$\mathtt{bot}!\mathtt{ide}? \sqcup^{\alpha} \mathtt{ide}?\mathtt{bot}! \;=\; \mathtt{ide}?\mathtt{ide}?.$$

To sidestep this problem we first introduce a normalform on contexts and restrict $\lesssim$ as well as the disjunction and conjunction to contexts in normalform. The definition of the normalform is straightforward: A subcontext which denotes the bottom projection is replaced by the canonical representation of $\perp\!\!\!\perp$. The normalform of both $\mathtt{bot}!\mathtt{ide}?$ and $\mathtt{ide}?\mathtt{bot}!$ is, of course, $\mathtt{bot}!\mathtt{bot}!$.

The function 'improper' defined below identifies bottom contexts. Note that the recursive context $\mu\beta.\kappa$ denotes $\perp\!\!\!\perp$ if and only if the first point of the corresponding Kleene chain, that is, $\kappa[\beta/\mathtt{bot}]$, evaluates to $\perp\!\!\!\perp$. Thus the context $\mu\beta.\mathtt{[]}\ \mathtt{bot} \mid \gamma?:\beta!$ represents the bottom projection. [A context on lists which does not accept the empty list may not at the same time specify a strict demand on the recursive component.]

6.13 Definition *The function* improper *identifies contexts which denote the bottom projection.*

$$\begin{array}{rcl} \mathrm{improper}[\![\alpha\kappa]\!]\varrho &=& \mathrm{norm}[\![\kappa]\!] = \mathtt{bot} \\ \mathrm{improper}[\![\beta]\!]\varrho &=& \varrho(\beta) \\ \mathrm{improper}[\![\varsigma\kappa]\!]\varrho &=& \kappa = \mathtt{bot} \\ \mathrm{improper}[\![\kappa\ell]\!]\varrho &=& \mathrm{improper}[\![\kappa]\!]\varrho \wedge \ell = ! \\ \mathrm{improper}[\![c_1\kappa_1 \mid \cdots \mid c_n\kappa_n]\!]\varrho &=& \mathrm{improper}[\![\kappa_1]\!]\varrho \wedge \cdots \wedge \mathrm{improper}[\![\kappa_n]\!]\varrho \\ \mathrm{improper}[\![\kappa_1 \ldots \kappa_n]\!]\varrho &=& \mathrm{improper}[\![\kappa_1]\!]\varrho \vee \cdots \vee \mathrm{improper}[\![\kappa_n]\!]\varrho \\ \mathrm{improper}[\![\mu\beta.\kappa]\!]\varrho &=& \mathrm{improper}[\![\kappa]\!](\varrho[\beta \mapsto \mathbf{t}]). \end{array}$$

Finally, $\text{improper}[\![\kappa]\!]$ *serves as an abbreviation for* $\text{improper}[\![\kappa]\!](\boldsymbol{\lambda}\beta.\mathbf{f})$.

6.14 Lemma *Let* $\kappa \in |\Gamma \vdash \sigma|$ *be a context. For all assignments* $\bar{\nu}$ *and* ϱ *with* $\bar{\nu}(\gamma_i) = [\,\mathbf{i}\,]$ *and* $\bar{\nu}(\beta) = \perp\!\!\!\perp \iff \varrho(\beta) = \mathbf{t}$ *the following holds,*

$$\mathcal{C}[\![\kappa]\!]\bar{\nu} = \perp\!\!\!\perp \quad\iff\quad \text{improper}[\![\kappa]\!]\varrho.$$

Proof By a straightforward induction over the type σ_0. Case $\sigma_0 = \alpha$: Holds by the Adequacy Theorem for polymorphic contexts. Case $\sigma_0 = \beta$: Immediate using the premise. Case $\sigma_0 = \mu\beta.\sigma$:

$$\begin{aligned}
\mathcal{C}[\![\mu\beta.\kappa]\!]\bar{\nu} = \perp\!\!\!\perp &\iff \mathcal{C}[\![\kappa]\!](\bar{\nu}[\beta \mapsto \perp\!\!\!\perp]) = \perp\!\!\!\perp && \text{property of the least fixpoint}\\
&\iff \text{improper}[\![\kappa]\!](\varrho[\beta \mapsto \mathbf{t}]) && \text{ind. hyp.}\\
&\iff \text{improper}[\![\mu\beta.\kappa]\!]\varrho && \text{def. improper}
\end{aligned}$$

The remaining cases are straightforward. ∎

Definition 6.15 introduces the canonical representations of the bottom context and the conjunctive unit, termed $\text{abs}(\sigma)$, which is employed in the definition of the guard operator.

6.15 Definition *Let* σ *be a type, then* $\text{bot}(\sigma)$ *denotes the bottom context on* σ.

$$\begin{aligned}
\text{bot}(\alpha) &= \alpha\texttt{bot}\\
\text{bot}(\beta) &= \beta\\
\text{bot}(\varsigma) &= \varsigma\texttt{bot}\\
\text{bot}(\sigma?) &= \text{bot}(\sigma)!\\
\text{bot}(c_1\sigma_1 \mid \cdots \mid c_n\sigma_n) &= c_1(\text{bot}(\sigma_1)) \mid \cdots \mid c_n(\text{bot}(\sigma_n))\\
\text{bot}(\sigma_1 \ldots \sigma_n) &= \text{bot}(\sigma_1) \ldots \text{bot}(\sigma_n)\\
\text{bot}(\mu\beta.\sigma) &= \mu\beta.\text{bot}(\sigma)
\end{aligned}$$

Accordingly, $\text{abs}(\sigma)$ *(pronounce 'absent') denotes the conjunctive unit on* σ.

$$\begin{aligned}
\text{abs}(\sigma?) &= \text{bot}(\sigma)?\\
\text{abs}(\varepsilon) &= \varepsilon\texttt{ide}\\
\text{abs}(\sigma_1 \ldots \sigma_n) &= \text{abs}(\sigma_1) \ldots \text{abs}(\sigma_n)
\end{aligned}$$

6.16 Fact *Let* σ *be a closed type, then*

1. $(\forall\nu)\,\mathcal{C}(\text{bot}(\sigma))\nu = \perp\!\!\!\perp$,
2. $(\forall\nu)\,\mathcal{C}(\text{abs}(\sigma))\nu = 1_{[\![\sigma]\!]}$.

Note that Fact 6.16 holds only for closed types since a free μ-bound type variable must not necessarily evaluate to $\perp\!\!\!\perp$.

Finally, Definition 6.17 introduces the normalform for contexts.

6.17 Definition [Extension of Definition 6.8] *The function* norm : $|\Gamma \vdash \sigma| \to |\Gamma \vdash \sigma|$ *maps a context to its normalform.*

$$\begin{array}{rcll}
\mathrm{norm}[\![\alpha\kappa]\!] & = & \alpha\mathrm{norm}[\![\kappa]\!] & \\
\mathrm{norm}[\![\beta]\!] & = & \beta & \\
\mathrm{norm}[\![\varsigma\kappa]\!] & = & \varsigma\kappa & \\
\mathrm{norm}[\![\kappa\ell]\!] & = & \mathrm{norm}[\![\kappa]\!]\ell & \\
\mathrm{norm}[\![c_1\kappa_1 \mid \cdots \mid c_n\kappa_n]\!] & = & c_1\,\mathrm{norm}[\![\kappa_1]\!] \mid \cdots \mid c_n\,\mathrm{norm}[\![\kappa_n]\!] & \\
\mathrm{norm}[\![\kappa_1 \ldots \kappa_n]\!] & = & \mathrm{bot}(\sigma_1 \ldots \sigma_n) & \textbf{if } \mathrm{improper}[\![\kappa_1 \ldots \kappa_n]\!] \\
& = & \mathrm{norm}[\![\kappa_1]\!] \ldots \mathrm{norm}[\![\kappa_n]\!] & \textbf{otherwise} \\
\mathrm{norm}(\mu\beta.\kappa) & = & \mathrm{bot}(\mu\beta.\sigma) & \textbf{if } \mathrm{improper}[\![\mu\beta.\kappa]\!] \\
& = & \mu\beta.\mathrm{norm}[\![\kappa]\!] & \textbf{otherwise}
\end{array}$$

The context κ *is called* in normalform *iff* $\mathrm{norm}[\![\kappa]\!] = \kappa$*. The set of all normal contexts on* σ *wrt* Γ *is denoted by* $\|\Gamma \vdash \sigma\|$.

Restricted to contexts in normalform the definition of $\lesssim$ is particularly simple. On base types we set `bot` $\lesssim$ `ide`; contexts on lifts, sequences etc are ordered coordinatewise. The relation $\sim$ even boils down to the identity relation (that is why it does not appear in the definition following).

6.18 Definition [Extension of Definition 6.4] *The relation* $\lesssim$ *on normal contexts is defined by the following axioms and rules.*

1. $\dfrac{\kappa \lesssim \kappa'}{\alpha\kappa \lesssim \alpha\kappa'}$ (α)

2. $\dfrac{}{\beta \lesssim \beta}$ (β)

3. $\dfrac{}{\varsigma\mathtt{bot} \lesssim \varsigma\kappa} \qquad \dfrac{}{\varsigma\kappa \lesssim \varsigma\mathtt{ide}}$ (ς)

4. $\dfrac{\kappa \lesssim \kappa'}{\kappa! \lesssim \kappa'\ell'} \qquad \dfrac{\kappa \lesssim \kappa'}{\kappa\ell \lesssim \kappa'?}$ $(\sigma?)$

5. $\dfrac{\kappa_1 \lesssim \kappa_1' \quad \ldots \quad \kappa_n \lesssim \kappa_n'}{c_1\kappa_1 \mid \cdots \mid c_n\kappa_n \lesssim c_1'\kappa_1' \mid \cdots \mid c_n'\kappa_n'}$ $(c_1\sigma_1 \mid \cdots \mid c_n\sigma_n)$

6. $\dfrac{\kappa_1 \lesssim \kappa_1' \quad \ldots \quad \kappa_n \lesssim \kappa_n'}{\kappa_1 \ldots \kappa_n \lesssim \kappa_1' \ldots \kappa_n'}$ $(\sigma_1 \ldots \sigma_n$ *with* $n \geqslant 1)$

7. $\dfrac{\kappa \lesssim \kappa'}{\mu\beta.\kappa \lesssim \mu\beta.\kappa'}$ $(\mu\beta.\sigma)$

The Adequacy Theorem establishes the equivalence of the semantic and syntactic notions of order and equality. As in the case of polymorphic contexts its proof employs the standard model (the interpretation of a context with respect to the assignment $\bar{\nu}(\gamma_i) = [\,\mathbf{i}\,]$) as the connecting link between syntax and semantics.

6.19 Adequacy Theorem *Let Γ be a finite set of assumptions and let $\kappa, \kappa' \in \|\Gamma \vdash \sigma\|$ be normal contexts. For all assignments $\bar\nu$ and $\bar\nu'$ with $\perp\!\!\!\perp \neq \bar\nu(\beta) \sqsubseteq \bar\nu'(\beta)$ and $\bar\nu(\gamma_i) = \bar\nu'(\gamma_i) = [\,\mathbf{i}\,]$ the following holds,*

$$\begin{array}{lcl} \kappa \approx \kappa' & \iff \mathcal{C}[\![\kappa]\!]\bar\nu = \mathcal{C}[\![\kappa']\!]\bar\nu \iff & \kappa = \kappa', \\ \kappa \lessapprox \kappa' & \iff \mathcal{C}[\![\kappa]\!]\bar\nu \sqsubseteq \mathcal{C}[\![\kappa']\!]\bar\nu' \iff & \kappa \lesssim \kappa'. \end{array}$$

Hence, $\langle \|\Gamma \vdash \sigma\|/\!\approx; \lessapprox\rangle$ and $\langle \|\Gamma \vdash \sigma\|; \lesssim\rangle$ are isomorphic finite partial orders.

Proof $(1) \Longrightarrow (2)$: Immediate since $\bar\nu$ and $\bar\nu'$ are assignments and $\mathcal{C}[\![\kappa]\!]$ is monotone. $(3) \Longrightarrow (1)$: It is not hard to see that the deduction rules are correct. $(2) \Longrightarrow (3)$: We show the proposition by induction over the type σ_0. Case $\sigma_0 = \alpha$: Follows by the Adequacy Theorem for polymorphic contexts. Case $\sigma_0 = \beta$ or $\sigma_0 = \varsigma$: Trivial. Case $\sigma_0 = \sigma?$:

$$\begin{array}{lll} & \mathcal{C}[\![\kappa\ell]\!]\bar\nu \sqsubseteq \mathcal{C}[\![\kappa'\ell']\!]\bar\nu' & \\ \Longrightarrow & \mathcal{C}[\![\kappa]\!]\bar\nu \sqsubseteq \mathcal{C}[\![\kappa']\!]\bar\nu' \wedge [\![\ell]\!] \sqsubseteq [\![\ell']\!] & \|D_\perp\| \cong \|D\| \times \mathbf{2} \\ \Longrightarrow & \kappa \lesssim \kappa' \wedge (\ell = {!} \vee \ell' = {?}) & \text{ind. hyp} \\ \Longrightarrow & \kappa\ell \lesssim \kappa'\ell' & \text{def.} \lesssim \end{array}$$

Case $\sigma_0 = c_1\sigma_1 \mid \cdots \mid c_n\sigma_n$: Analogous. Case $\sigma_0 = \sigma_1 \ldots \sigma_n$: The proposition holds trivially if improper$[\![\kappa_1 \ldots \kappa_n]\!]$. Hence, assume that $\neg$improper$[\![\kappa_1 \ldots \kappa_n]\!]$ which in turn implies $(\forall\nu)\, \mathcal{C}[\![\kappa_i]\!]\bar\nu \neq \perp\!\!\!\perp$ by Lemma 6.14.

$$\begin{array}{lll} & \mathcal{C}[\![\kappa_1 \ldots \kappa_n]\!]\bar\nu \sqsubseteq \mathcal{C}[\![\kappa'_1 \ldots \kappa'_n]\!]\bar\nu' & \lceil\ \|D_1 \otimes \cdots \otimes D_n\| \cong \|D_1\| \otimes \cdots \otimes \|D_n\| \\ \Longrightarrow & \langle \mathcal{C}[\![\kappa_1]\!]\bar\nu, \ldots, \mathcal{C}[\![\kappa_n]\!]\bar\nu\rangle \sqsubseteq \langle \mathcal{C}[\![\kappa'_1]\!]\bar\nu', \ldots, \mathcal{C}[\![\kappa'_n]\!]\bar\nu'\rangle & \\ \Longrightarrow & (\forall i)\ \mathcal{C}[\![\kappa_i]\!]\bar\nu \sqsubseteq \mathcal{C}[\![\kappa'_i]\!]\bar\nu' & (\forall i)\ \mathcal{C}[\![\kappa_i]\!]\bar\nu \neq \perp\!\!\!\perp \\ \Longrightarrow & (\forall i)\ \kappa_i \lesssim \kappa'_i & \text{ind. hyp} \\ \Longrightarrow & \kappa_1 \ldots \kappa_n \lesssim \kappa'_1 \ldots \kappa'_n & \text{def.} \lesssim \end{array}$$

Case $\sigma_0 = \mu\beta.\sigma$: Let $\pi = \mathcal{C}[\![\mu\beta.\kappa]\!]\bar\nu$ and $\pi' = \mathcal{C}[\![\mu\beta.\kappa']\!]\bar\nu'$. Without loss of generality we assume that $\neg$improper$[\![\mu\beta.\kappa]\!]$ which implies $\pi \neq \perp\!\!\!\perp$ by Lemma 6.14.

$$\begin{array}{lll} & \mathcal{C}[\![\mu\beta.\kappa]\!]\bar\nu \sqsubseteq \mathcal{C}[\![\mu\beta.\kappa']\!]\bar\nu' & \\ \Longrightarrow & \mathcal{C}[\![\kappa]\!](\bar\nu[\beta \mapsto \pi]) \sqsubseteq \mathcal{C}[\![\kappa']\!](\bar\nu'[\beta \mapsto \pi']) & \mathsf{\textbf{lfp}}\,\varphi = \varphi(\mathsf{\textbf{lfp}}\,\varphi) \\ \Longrightarrow & \kappa \lesssim \kappa' & \text{ind. hyp} \\ \Longrightarrow & \mu\beta.\kappa \lesssim \mu\beta.\kappa' & \text{def.} \lesssim \end{array}$$

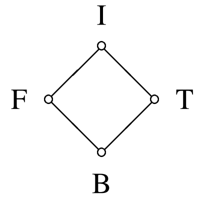
Finally, $\mathcal{C}[\![\kappa]\!]\bar\nu = \mathcal{C}[\![\kappa']\!]\bar\nu$ implies $\mathcal{C}[\![\kappa]\!]\bar\nu \sqsubseteq \mathcal{C}[\![\kappa']\!]\bar\nu$ and $\mathcal{C}[\![\kappa']\!]\bar\nu \sqsubseteq \mathcal{C}[\![\kappa]\!]\bar\nu$. By the first part we have $\kappa \lesssim \kappa'$ and $\kappa' \lesssim \kappa$ which yields $\kappa = \kappa'$. ■

Let us consider some examples for context orders (well, contexts form in fact finite lattices as we shall see in the next section.)

Truth values The set of contexts $\|$`bool`$\|$ contains four different elements, namely

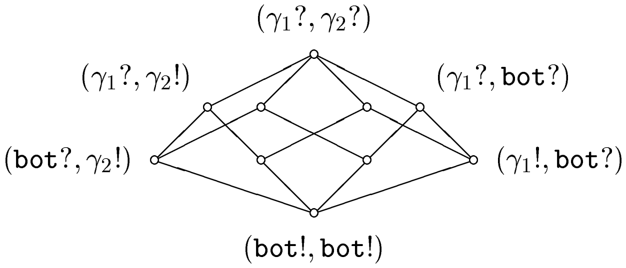
$$\begin{array}{lclclcl} \mathrm{B} & = & \texttt{False bot} \mid \texttt{True bot} & \quad & \mathrm{F} & = & \texttt{False ide} \mid \texttt{True bot} \\ \mathrm{I} & = & \texttt{False ide} \mid \texttt{True ide} & & \mathrm{T} & = & \texttt{False bot} \mid \texttt{True ide}. \end{array}$$

Apart from bottom and identity we have the context F which accepts only `False` and its counterpart T which accepts only `True`. It is immediate that $\|$`bool`$\|$ is isomorphic to $\mathbf{2} \times \mathbf{2}$.



Note that this context lattice involves no approximations since it is isomorphic to the corresponding projection lattice (cf Section 4.3). This property holds generally for enumeration types.

Polymorphic pairs Data types consisting only of one constructor are commonly called *tuple types*. Polymorphic pairs, for example, are defined by $\texttt{(}\alpha_1\texttt{,}\alpha_2\texttt{)}\ \texttt{::=}\ \texttt{(}\alpha_1\texttt{,}\alpha_2\texttt{)}$. Since the right hand side of the definition consists of a one element sum, pairs are 'isomorphic' to sequences of length two. Let $\Gamma = \{\gamma_1 : \|\alpha_1\|, \gamma_2 : \|\alpha_2\|\}$, then $\|\Gamma \vdash (\alpha_1, \alpha_2)\|$ has the following shape.



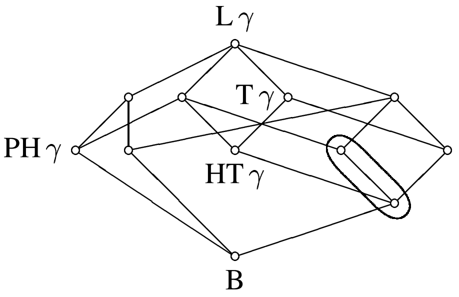
Since we confine ourselves to independent projections respective contexts and bot and γ_i are the only contexts on α_i, the context lattice $\|\Gamma \vdash (\alpha_1, \alpha_2)\|$ is in fact isomorphic to $\mathbf{i}\,\|\mathbf{2}_\perp \otimes \mathbf{2}_\perp\|$ (cf to Section 4.4).

Polymorphic lists The polymorphic list type, $\texttt{[}\alpha\texttt{]} = \mu\beta.\texttt{[]}\ \varepsilon \mid \alpha?\texttt{:}\beta?$, gives rise to 11 different normal contexts—provided we restrict the polymorphic contexts on α to bot and γ. The following abbreviations are used in the sequel.

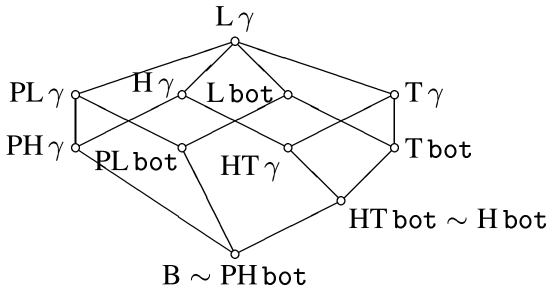
$$\begin{array}{rcl@{\qquad}rcl}
\mathrm{B} & = & \mu\beta.\texttt{[] bot} \mid \texttt{bot!:}\beta! & \mathrm{HT}\,\gamma & = & \mu\beta.\texttt{[] ide} \mid \gamma!\texttt{:}\beta! \\
\mathrm{PH}\,\gamma & = & \mu\beta.\texttt{[] bot} \mid \gamma!\texttt{:}\beta? & \mathrm{H}\,\gamma & = & \mu\beta.\texttt{[] ide} \mid \gamma!\texttt{:}\beta? \\
\mathrm{PL}\,\gamma & = & \mu\beta.\texttt{[] bot} \mid \gamma?\texttt{:}\beta? & \mathrm{T}\,\gamma & = & \mu\beta.\texttt{[] ide} \mid \gamma?\texttt{:}\beta! \\
 & & & \mathrm{L}\,\gamma & = & \mu\beta.\texttt{[] ide} \mid \gamma?\texttt{:}\beta?
\end{array}$$

The letters stand for: 'B' like bottom, 'P' like partial and infinite lists only (`[]` is not admissible), 'H' like head strict, 'T' like tail strict, and 'L' like identity on lists. Monomorphic variants of HT γ, H γ, T γ, and L γ have already been introduced in Chapter 1. The contexts PH γ and PL γ correspond to H γ and L γ with the notable exception that they do not accept the empty list.

Let us construct the lattice $\|\Gamma \vdash \texttt{[}\alpha\texttt{]}\|$ with $\Gamma = \{\gamma : \|\alpha\|\}$ systematically. The prototype context on polymorphic lists is $\mu\beta.\texttt{[]}\ \kappa_1 \mid \kappa_2\ell_1\texttt{:}\beta\ell_2$ with $\kappa_1 \in \{\texttt{bot}, \texttt{ide}\}$, $\kappa_2 \in \{\texttt{bot}, \gamma\}$ and $\ell_1, \ell_2 \in \{!, ?\}$. Case $\kappa_1 = \texttt{bot}$: The prototype reduces to B if and only if either $\kappa_2 = \texttt{bot}$ and $\ell_1 = !$ or else $\ell_2 = !$. The remaining combinations, namely PH γ, PL bot, and PL γ are ordered by PH $\gamma \lesssim$ PL γ and PL bot $\lesssim$ PL γ. Case $\kappa_1 = \texttt{ide}$: In principal there are eight different combinations but two of them coincide, namely HT bot $\sim$ H bot. We obtain a diagram for the context lattice by placing a copy of $\mathbf{2}^2$ (first case) and a copy of $\mathbf{2}^3$ (second case with HT bot and H bot identified) side by side connecting corresponding points.



From this draft we obtain the following more symmetric diagram simply by flipping L `bot` and T γ.



Polymorphic trees As a final example consider the type of polymorphic trees given by tree $\alpha = \mu\beta$.Empty ε | Node $\beta?\ \alpha?\ \beta?$. Let us again introduce some abbreviations.

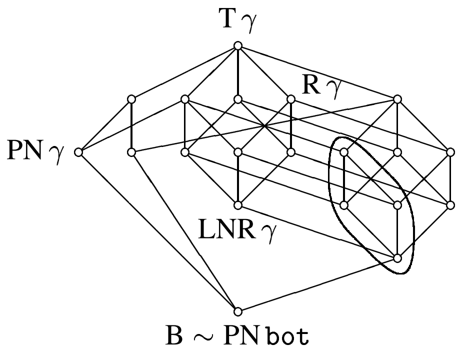
$$\begin{aligned}
\mathrm{B} &= \mu\beta.\mathrm{Empty}\ \texttt{bot} \mid \mathrm{Node}\ \beta!\ \texttt{bot}!\ \beta! \\
\mathrm{PN}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{bot} \mid \mathrm{Node}\ \beta?\ \gamma!\ \beta? \\
\mathrm{PT}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{bot} \mid \mathrm{Node}\ \beta?\ \gamma?\ \beta?
\end{aligned}$$

The mnemonic system is similar to the one for polymorphic lists. The letter 'B' represents the bottom context, 'P' comprises partial and infinite trees (Empty is not admissible). The three lifts of the constructor 'Node' are coded into the letters 'L', 'N', and 'R' respectively which stand for strict in the left subtree, strict in the element labelled to the node and strict in the right subtree. Finally, 'T' represents the identity context on trees.

$$\begin{aligned}
\mathrm{LNR}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{ide} \mid \mathrm{Node}\ \beta!\ \gamma!\ \beta! \\
\mathrm{LN}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{ide} \mid \mathrm{Node}\ \beta!\ \gamma!\ \beta? \\
\mathrm{LR}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{ide} \mid \mathrm{Node}\ \beta!\ \gamma?\ \beta! \\
\mathrm{L}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{ide} \mid \mathrm{Node}\ \beta!\ \gamma?\ \beta? \\
\mathrm{NR}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{ide} \mid \mathrm{Node}\ \beta?\ \gamma!\ \beta! \\
\mathrm{N}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{ide} \mid \mathrm{Node}\ \beta?\ \gamma!\ \beta? \\
\mathrm{R}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{ide} \mid \mathrm{Node}\ \beta?\ \gamma?\ \beta! \\
\mathrm{T}\,\gamma &= \mu\beta.\mathrm{Empty}\ \texttt{ide} \mid \mathrm{Node}\ \beta?\ \gamma?\ \beta?
\end{aligned}$$

Caveat: We have overloaded the letters 'L' and 'T' which abbreviate different contexts depending on their types. However, this should cause no problems since the type is usually known.

The diagram for $\|\Gamma \vdash \mathrm{tree}\,\alpha\|$ may be obtained in a similar fashion as the one for polymorphic lists. Since the constructor 'Node' contains four variable parts (three lifts plus a polymorphic context) we start by placing copies of $\mathbf{2}^2$ and $\mathbf{2}^4$ side by side.



Note that we have surrounded the points which possess the same normalform. If the context on α is `bot` and its lift is strict the context $\mu\beta$.Empty `ide` $\mid$ Node $\beta\ell_1$ `bot`! $\beta\ell_2$ collapses to LNR `bot` irrespective of the lifts ℓ_1 and ℓ_2. The context LNR `bot` corresponds to HT `bot` since it accepts only the empty tree.

6.4 Disjunction and conjunction

Having constructed abstract domains of contexts we must transfer the operators $\&$, $\sqcup$, $\rhd$ etc to these domains. This section deals with the conjunction and disjunction of demands, the rest of the operators are introduced in Chapter 7. There are three desiderata on the abstract variant of an operator:

1. The result of the operator must be in normalform—provided it is a context.

2. The operator must be monotone with respect to $\lesssim$ in order to guarantee the existence of least fixpoints.

3. The operator must be safe with respect to the concrete one. If $\odot^\alpha$ is the abstract variant of $\odot$, then $\odot^\alpha$ is safe with respect to $\odot$ iff for all normal contexts $\kappa, \kappa' \in \|\Gamma \vdash \sigma\|$,

$$(\forall\nu)\ \mathcal{C}[\![\kappa]\!]\nu \odot \mathcal{C}[\![\kappa']\!]\nu \sqsubseteq \mathcal{C}(\kappa \odot^\alpha \kappa')\nu.$$

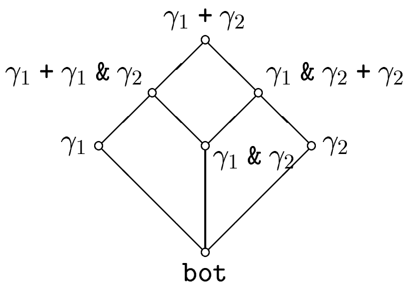
   Analogous for unary or multiary operators.

Since $\lesssim$ amounts to the coordinatewise order, the join of contexts is formed coordinatewise as well, for example, on recursive contexts we have

$$(\mu\beta.\kappa) \sqcup^\alpha (\mu\beta.\kappa') \quad = \quad \mu\beta.\kappa \sqcup^\alpha \kappa'.$$

Most of the laws concerning the conjunction of demands have already been introduced in Lemma 4.25. It remains to derive a suitable law for the conjunction of recursive contexts. Contrary to the disjunction the conjunction may not be pushed inwards. As a rule the context $\mu\beta.\kappa \mathbin{\&^\alpha} \kappa'$ is strictly less than $(\mu\beta.\kappa) \mathbin{\&^\alpha} (\mu\beta.\kappa')$, for example, HT$\gamma \lnsim$ H$\gamma \mathbin{\&^\alpha}$ Tγ. Equality only holds if the conjunction propagates to each of the recursive calls.

Hence we are forced to determine the recursive calls in $\kappa \mathbin{\&^\alpha} \kappa'$. Treating β in κ and κ' as different generic type variables, say, γ_1 and γ_2 there are seven possible outcomes (cf to Section 6.1), which are ordered as follows.



Uniformity requires us to take the least upper bound of the recursive calls and $\gamma_1 \,\&\, \gamma_2$ itself as the template for the conjunction. If for example the least upper bound yields $\gamma_1 + \gamma_1 \,\&\, \gamma_2$, the conjunction is *at least* $\mu\beta.\kappa \sqcup^\alpha \kappa \mathbin{\&}^\alpha \kappa'$. The question is whether this approximation is safe: If we repeated the procedure with $\kappa \sqcup^\alpha \kappa \mathbin{\&}^\alpha \kappa'$, could it be that the result is strictly greater than $\gamma_1 + \gamma_1 \,\&\, \gamma_2$. Fortunately, the second round always produces the same template. This phenomenon is readily explained: Adjoining $\kappa \sqcup^\alpha \cdot$ to $\kappa \mathbin{\&}^\alpha \kappa'$ can only result in adjoining $\gamma_1 \,\&\, \cdot$ or $\gamma_1 + \cdot$ to $\gamma_1 + \gamma_1 \,\&\, \gamma_2$. Both operations leave the template unchanged.

$$\begin{aligned} \gamma_1 \,\&\, (\gamma_1 + \gamma_1 \,\&\, \gamma_2) &= \gamma_1 \,\&\, \gamma_1 + \gamma_1 \,\&\, \gamma_1 \,\&\, \gamma_2 = \gamma_1 + \gamma_1 \,\&\, \gamma_2 \\ \gamma_1 + (\gamma_1 + \gamma_1 \,\&\, \gamma_2) &= \gamma_1 + \gamma_1 \,\&\, \gamma_2 \end{aligned}$$

This property holds trivially if the first round yields $\gamma_1 \,\&\, \gamma_2$ (ie, a fixpont is reached) or $\gamma_1 + \gamma_2$ (it cannot get worse).

Consequently we may indeed take the least upper bound of the recursive calls and $\gamma_1 \,\&\, \gamma_2$ as a template for the conjunction of recursive contexts. If $\gamma_1 + \gamma_1 \,\&\, \gamma_2$ is the least upper bound, we set $(\mu\beta.\kappa) \mathbin{\&}^\alpha (\mu\beta.\kappa') = \mu\beta.\kappa \sqcup^\alpha \kappa \mathbin{\&}^\alpha \kappa'$. The following table shows the effect of $\mathbin{\&}^\alpha$ to the prototype contexts on lists.

&	Bγ	PHγ	PLγ	HTγ	Hγ	Tγ	Lγ
Bγ	Bγ	Bγ	Bγ	Bγ	Bγ	Bγ	Bγ
PHγ		PHγ	PLγ	HTγ	Hγ	Tγ	Lγ
PLγ			PLγ	HTγ	Lγ	Tγ	Lγ
HTγ				HTγ	HTγ	HTγ	HTγ
Hγ					Hγ	Tγ	Lγ
Tγ						Tγ	Tγ
Lγ							Lγ

The operation which determines the least upper bound of the recursive calls is formalized in the subsequent definition.

6.20 Definition *Let $\kappa \in \|\Gamma \vdash \sigma\|$ be a normal context and let $\alpha \in$ **tvar** be a generic type variable. Then, $\kappa \updownarrow \alpha$ calculates the least upper bound of the polymorphic contexts on α which appear as subcontexts in κ.*

$$\begin{aligned} \alpha'\kappa \updownarrow \alpha &= \kappa && \textbf{if } \alpha = \alpha' \\ &= \texttt{bot} && \textbf{otherwise} \\ \beta \updownarrow \alpha &= \texttt{bot} \\ \varsigma\kappa \updownarrow \alpha &= \texttt{bot} \\ \kappa\ell \updownarrow \alpha &= \kappa \updownarrow \alpha \\ (c_1\kappa_1 \mid \cdots \mid c_n\kappa_n) \updownarrow \alpha &= \kappa_1 \updownarrow \alpha \sqcup^\alpha \cdots \sqcup^\alpha \kappa_n \updownarrow \alpha \\ (\kappa_1 \ldots \kappa_n) \updownarrow \alpha &= \kappa_1 \updownarrow \alpha \sqcup^\alpha \cdots \sqcup^\alpha \kappa_n \updownarrow \alpha \\ \mu\beta.\kappa \updownarrow \alpha &= \kappa \updownarrow \alpha \end{aligned}$$

Definition 6.21 summarizes the equations defining the conjunction and disjunction of normal contexts. Recall from the desiderata listed at the beginning of this section that we have to take care that the result is again in normalform. This condition is automatically satisfied for the disjunction of contexts since $\kappa_1 \sqcup^\alpha \kappa_2$ is greater than both κ_1 and κ_2. Contrary to the disjunction the conjunction of contexts may yield bottom even if neither of its arguments is the bottom context, for instance F $\mathbin{\&}^\alpha$ T $=$ B (recall that F, T, B are contexts on `bool`). Consequently we must normalize conjuncts of sequence and recursive types. Using 'F' and 'T' it is relatively easy to find examples which demonstrate the necessity to do so:

$$(\text{F! I?}) \mathbin{\&}^\alpha (\text{T! I?}) \quad = \quad \text{norm}[\![\text{B! I?}]\!] \quad = \quad \text{B! B!},$$

$$(\mu\beta.\text{Leaf F!} \mid \text{Node } \beta!\ \text{I?}\ \beta!) \mathbin{\&}^\alpha (\mu\beta.\text{Leaf T!} \mid \text{Node } \beta!\ \text{I?}\ \beta!)$$
$$= \quad \text{norm}[\![\mu\beta.\text{Leaf B!} \mid \text{Node } \beta!\ \text{I?}\ \beta!]\!] \quad = \quad \mu\beta.\text{Leaf B!} \mid \text{Node } \beta!\ \text{B!}\ \beta!.$$

In the last example we have employed contexts on the data type of binary trees with Boolean labels $\mu\beta.$Leaf bool? $\mid$ Node β? bool? β?.

6.21 Definition [Extension of Definition 6.8] *Conjunction and disjunction of normal contexts are defined as follows.*

1. $\alpha\kappa \sqcup^\alpha \alpha\kappa' \;=\; \alpha(\kappa \sqcup^\alpha \kappa') \qquad \alpha\kappa \mathbin{\&}^\alpha \alpha\kappa' \;=\; \alpha(\kappa \mathbin{\&}^\alpha \kappa')$ *(α)*

2. $\beta \sqcup^\alpha \beta \;=\; \beta \qquad \beta \mathbin{\&}^\alpha \beta \;=\; \beta$ *(β)*

3. *(ς)*
$$\begin{aligned} \varsigma\texttt{bot} \sqcup^\alpha \varsigma\kappa &= \varsigma\kappa & \varsigma\texttt{bot} \mathbin{\&}^\alpha \varsigma\kappa &= \varsigma\texttt{bot} \\ \varsigma\kappa \sqcup^\alpha \varsigma\texttt{bot} &= \varsigma\kappa & \varsigma\kappa \mathbin{\&}^\alpha \varsigma\texttt{bot} &= \varsigma\texttt{bot} \\ \varsigma\texttt{ide} \sqcup^\alpha \varsigma\texttt{ide} &= \varsigma\texttt{ide} & \varsigma\texttt{ide} \mathbin{\&}^\alpha \varsigma\texttt{ide} &= \varsigma\texttt{ide} \end{aligned}$$

4. *(σ?)*
$$\begin{aligned} \kappa! \sqcup^\alpha \kappa'! &= (\kappa \sqcup^\alpha \kappa')! & \kappa! \mathbin{\&}^\alpha \kappa'! &= (\kappa \mathbin{\&}^\alpha \kappa')! \\ \kappa! \sqcup^\alpha \kappa'? &= (\kappa \sqcup^\alpha \kappa')? & \kappa! \mathbin{\&}^\alpha \kappa'? &= (\kappa \sqcup^\alpha \kappa \mathbin{\&}^\alpha \kappa')! \\ \kappa? \sqcup^\alpha \kappa'! &= (\kappa \sqcup^\alpha \kappa')? & \kappa? \mathbin{\&}^\alpha \kappa'! &= (\kappa \mathbin{\&}^\alpha \kappa' \sqcup^\alpha \kappa')! \\ \kappa? \sqcup^\alpha \kappa'? &= (\kappa \sqcup^\alpha \kappa')? & \kappa? \mathbin{\&}^\alpha \kappa'? &= (\kappa \sqcup^\alpha \kappa')? \end{aligned}$$

5.
$$\begin{aligned} (c_1\kappa_1 \mid \cdots \mid c_n\kappa_n) \sqcup^\alpha (c_1\kappa_1' \mid \cdots \mid c_n\kappa_n') &= c_1(\kappa_1 \sqcup^\alpha \kappa_1') \mid \cdots \mid c_n(\kappa_n \sqcup^\alpha \kappa_n') \\ (c_1\kappa_1 \mid \cdots \mid c_n\kappa_n) \mathbin{\&}^\alpha (c_1\kappa_1' \mid \cdots \mid c_n\kappa_n') &= c_1(\kappa_1 \mathbin{\&}^\alpha \kappa_1') \mid \cdots \mid c_n(\kappa_n \mathbin{\&}^\alpha \kappa_n') \end{aligned}$$
$\lfloor$ ($c_1\sigma_1 \mid \cdots \mid c_n\sigma_n$)

6. *($\sigma_1 \ldots \sigma_n$)*
$$\begin{aligned} (\kappa_1 \ldots \kappa_n) \sqcup^\alpha (\kappa_1' \ldots \kappa_n') &= (\kappa_1 \sqcup^\alpha \kappa_1') \ldots (\kappa_n \sqcup^\alpha \kappa_n') \\ (\kappa_1 \ldots \kappa_n) \mathbin{\&}^\alpha (\kappa_1' \ldots \kappa_n') &= \text{norm}((\kappa_1 \mathbin{\&}^\alpha \kappa_1') \ldots (\kappa_n \mathbin{\&}^\alpha \kappa_n')) \end{aligned}$$

7. *($\mu\beta.\sigma$)*
$$\begin{aligned} (\mu\beta.\kappa) \sqcup^\alpha (\mu\beta.\kappa') &= \mu\beta.\kappa \sqcup^\alpha \kappa' & & \\ (\mu\beta.\kappa) \mathbin{\&}^\alpha (\mu\beta.\kappa') &= \text{norm}(\mu\beta.\kappa \mathbin{\&}^\alpha \kappa') & &\textbf{if } \bar{\kappa} \lesssim \gamma_1 \,\&\, \gamma_2 \\ &= \mu\beta.\kappa \sqcup^\alpha \kappa \mathbin{\&}^\alpha \kappa' & &\textbf{if } \bar{\kappa} \lesssim \gamma_1 + \gamma_1 \,\&\, \gamma_2 \\ &= \mu\beta.\kappa \mathbin{\&}^\alpha \kappa \sqcup^\alpha \kappa' & &\textbf{if } \bar{\kappa} \lesssim \gamma_1 \,\&\, \gamma_2 + \gamma_2 \\ &= \mu\beta.\kappa \sqcup^\alpha \kappa' & &\textbf{if } \bar{\kappa} \lesssim \gamma_1 + \gamma_2 \end{aligned}$$
where
$$\bar{\kappa} = (\text{norm}(\kappa[\beta/\alpha\gamma_1]) \mathbin{\&}^\alpha \text{norm}(\kappa[\beta/\alpha\gamma_2])) \updownarrow \alpha$$

It is immediate that the disjunction of contexts corresponds to the join operation. Since furthermore bot(α) is the least element and $\|\Gamma \vdash \sigma\|$ is finite—provided Γ is—we have the following

6.22 Fact *Let Γ be a finite set of assumptions and let $\kappa, \kappa' \in \|\Gamma \vdash \sigma\|$ be normal contexts. Then $\kappa \sqcup^\alpha \kappa'$ is the join of κ and κ' and furthermore $\langle \|\Gamma \vdash \sigma\|; \lesssim \rangle$ forms a finite lattice.*

It remains to show that both $\&^\alpha$ and $\sqcup^\alpha$ are monotone and safe. The safety of $\sqcup^\alpha$ with respect to $\sqcup$ is an immediate consequence of the Adequacy Theorem.

6.23 Corollary *The disjunction of contexts $\sqcup^\alpha$ is monotone and a safe approximation of $\sqcup$, that is, for all normal contexts $\kappa, \kappa' \in \|\Gamma \vdash \sigma\|$,*

$$(\forall \nu)\ \mathcal{C}[\![\kappa]\!]\nu \sqcup \mathcal{C}[\![\kappa']\!]\nu \sqsubseteq \mathcal{C}(\kappa \sqcup^\alpha \kappa')\nu.$$

Furthermore, $\sqcup^\alpha$ is the best approximation of $\sqcup$.

Proof Monotonicity is implied by the properties of the join. The safety *and* the optimality of $\sqcup^\alpha$ follow by the following calculation (the safety follows with $\kappa := \kappa_1 \sqcup^\alpha \kappa_2$).

$$\begin{array}{lll} & (\forall \nu)\ \mathcal{C}[\![\kappa_1]\!]\nu \sqcup \mathcal{C}[\![\kappa_2]\!]\nu \sqsubseteq \mathcal{C}[\![\kappa]\!]\nu & \\ \iff & \kappa_1 \lessapprox \kappa \wedge \kappa_2 \lessapprox \kappa & \text{property of } \sqcup \text{ and def. } \lessapprox \\ \iff & \kappa_1 \lesssim \kappa \wedge \kappa_2 \lesssim \kappa & \text{Adequacy Theorem} \\ \iff & \kappa_1 \sqcup^\alpha \kappa_2 \lesssim \kappa & \text{property of } \sqcup^\alpha \ \blacksquare \end{array}$$

Note that the proof of safety is based on the soundness part of the Adequacy Theorem while the optimality of $\sqcup^\alpha$ rests on the completeness part.

6.24 Lemma *The conjunction of contexts $\&^\alpha$ is monotone and a safe approximation of $\&$, that is, for all normal contexts $\kappa, \kappa' \in \|\Gamma \vdash \sigma\|$,*

$$(\forall \nu)\ \mathcal{C}[\![\kappa]\!]\nu \mathbin{\&} \mathcal{C}[\![\kappa']\!]\nu \sqsubseteq \mathcal{C}(\kappa \mathbin{\&^\alpha} \kappa')\nu.$$

Proof Monotonicity follows by a straightforward induction over the type using $\kappa \mathbin{\&^\alpha} \kappa' \lesssim \kappa \sqcup^\alpha \kappa'$. We show the safety of $\&^\alpha$ by induction over the size of σ_0. Case $\sigma_0 = \alpha$: We have $\mathcal{C}[\![\mathcal{A}]\!]\nu \mathbin{\&} \mathcal{C}[\![\mathcal{A}']\!]\nu = \mathcal{C}(\mathcal{A} \mathbin{\&^\alpha} \mathcal{A}')\nu$. Case $\sigma_0 = \beta$ or $\sigma_0 = \varsigma$: Immediate. Case $\sigma_0 = \sigma?$:

$$\begin{array}{lll} & \mathcal{C}[\![\kappa!]\!]\nu \mathbin{\&} \mathcal{C}[\![\kappa'?]\!]\nu & \\ = & (\mathcal{C}[\![\kappa]\!]\nu \sqcup \mathcal{C}[\![\kappa]\!]\nu \mathbin{\&} \mathcal{C}[\![\kappa']\!]\nu)_\oplus & \text{def. } \mathcal{C} \text{ and Lemma 4.25} \\ \sqsubseteq & (\mathcal{C}(\kappa \sqcup^\alpha \kappa \mathbin{\&^\alpha} \kappa')\nu)_\oplus & \text{ind. hyp} \\ = & \mathcal{C}(\kappa! \mathbin{\&^\alpha} \kappa'?)\nu & \text{def. } \mathcal{C} \text{ and } \&^\alpha \end{array}$$

The remaining cases are shown accordingly. Case $\sigma_0 = c_1\sigma_1 \mid \cdots \mid c_n\sigma_n$ or $\sigma_0 = \sigma_1 \ldots \sigma_n$: Analogous. Case $\sigma_0 = \mu\beta.\sigma$: Assume that $\bar{\kappa} = \gamma_1 + \gamma_1 \mathbin{\&} \gamma_2$. Let $\kappa_1 = \kappa[\beta/\alpha\gamma_1]$, $\kappa_2 = \kappa[\beta/\alpha\gamma_2]$ and $\hat{\kappa} = (\kappa \sqcup^\alpha \kappa \mathbin{\&^\alpha} \kappa')[\beta/\alpha\bar{\kappa}]$. Clearly, $\kappa_1 \sqcup^\alpha \kappa_1 \mathbin{\&^\alpha} \kappa_2 \lesssim \hat{\kappa}$ since both sides differ only in the 'recursive calls' and $\bar{\kappa}$ is their upper bound. We show

$$\mathcal{C}[\![\mu\beta.\kappa]\!]\nu \sqcup \mathcal{C}[\![\mu\beta.\kappa]\!]\nu \mathbin{\&} \mathcal{C}[\![\mu\beta.\kappa']\!]\nu \sqsubseteq \mathcal{C}[\![\mu\beta.\hat{\kappa}]\!]\nu$$

by Fixpoint Induction which implies the proposition. Note that the predicate $P(\pi, \pi', \hat{\pi}) \iff \pi \sqcup \pi \mathbin{\&} \pi' \sqsubseteq \hat{\pi}$ is admissible since $\sqcup$ and $\&$ are continuous. Induction basis: Trivial. Induction step: Assume that $P(\pi, \pi', \hat{\pi})$ holds. Let $\bar{\nu} = \nu[\gamma_1 \mapsto \pi][\gamma_2 \mapsto \pi']$, then

$$\begin{array}{lll} & \mathcal{C}[\![\kappa]\!](\nu[\beta \mapsto \pi]) \sqcup \mathcal{C}[\![\kappa]\!](\nu[\beta \mapsto \pi]) \,\&\, \mathcal{C}[\![\kappa']\!](\nu[\beta \mapsto \pi']) & \\ = & \mathcal{C}[\![\kappa_1]\!]\bar{\nu} \sqcup \mathcal{C}[\![\kappa_1]\!]\bar{\nu} \sqcup \&\mathcal{C}[\![\kappa_2]\!]\bar{\nu} & \text{Substitution Lemma} \\ \sqsubseteq & \mathcal{C}(\kappa_1 \sqcup^\alpha \kappa_1 \,\&^\alpha\, \kappa_2)\bar{\nu} & \text{outer ind. hyp.} \\ \sqsubseteq & \mathcal{C}[\![\hat{\kappa}]\!]\bar{\nu} & \kappa_1 \sqcup^\alpha \kappa_1 \,\&^\alpha\, \kappa_2 \lesssim \hat{\kappa} \text{ and Adequacy Theorem} \\ = & \mathcal{C}(\kappa \sqcup^\alpha \kappa \,\&^\alpha\, \kappa')(\nu[\beta \mapsto \pi \sqcup \pi \,\&\, \pi']) & \text{Substitution Lemma} \\ \sqsubseteq & \mathcal{C}(\kappa \sqcup^\alpha \kappa \,\&^\alpha\, \kappa')(\nu[\beta \mapsto \hat{\pi}]) & \text{inner ind. hyp.} \end{array}$$

The remaining cases are treated in a similar fashion. ∎

Unfortunately, it is not known whether $\&^\alpha$ is the best approximation of $\&$.

Having defined abstract variants of $\&$ and $\sqcup$ it is natural to ask whether the properties of $\&$ and $\sqcup$ listed in Lemma 4.23 remain valid. It is immediate that both operations are associative, commutative and idempotent. Furthermore, $\text{bot}(\sigma)$ is the zero element of $\&^\alpha$ and the unit of $\sqcup^\alpha$; $\text{abs}(\sigma)$ is the unit of $\&^\alpha$. The inclusion law holds since $\kappa_1 \,\&^\alpha\, \kappa_2 \lesssim \kappa_1 \sqcup^\alpha \kappa_2$. However, the distributive laws fail! This failure is caused by the restriction to independent projections as the following example shows.

$$\begin{array}{lllll} \text{F! T!} \,\&^\alpha\, (\text{F! F!} \sqcup^\alpha \text{T! T!}) & = & \text{F! T!} \,\&^\alpha\, \text{I! I!} & = & \text{F! T!} \\ \text{F! T!} \,\&^\alpha\, \text{F! F!} \sqcup^\alpha \text{F! T!} \,\&^\alpha\, \text{T! T!} & = & \text{B! B!} \sqcup^\alpha \text{B! B!} & = & \text{B! B!}. \end{array}$$

Generally, we only have $\kappa_1 \,\&^\alpha\, (\kappa_2 \sqcup^\alpha \kappa_3) \gtrsim \kappa_1 \,\&^\alpha\, \kappa_2 \sqcup^\alpha \kappa_1 \,\&^\alpha\, \kappa_3$.

We conclude the section with estimations of the size and the length of context lattices. We have already indicated that the length attributes to an upper bound for the number of fixpoint iterations. The cardinality of context lattices determines the size of context transformers which in turn greatly influences the effectivity of the strictness analyser. The following definition introduces a norm for contexts.

6.25 Definition *The function* size *is given by*

$$\begin{array}{rcl} \text{size}(\Gamma \vdash \alpha) & = & 2^{\,\text{card}(\Gamma\restriction_\alpha)} - 1 \\ \text{size}(\Gamma \vdash \beta) & = & 1 \\ \text{size}(\Gamma \vdash \varsigma) & = & 1 \\ \text{size}(\Gamma \vdash \sigma?) & = & \text{size}(\Gamma \vdash \sigma) + 1 \\ \text{size}(\Gamma \vdash \sigma_1 \ldots \sigma_n) & = & \text{size}(\Gamma \vdash \sigma_1 + \cdots + (\text{size}(\Gamma \vdash \sigma_n) \\ \text{size}(\Gamma \vdash c_1\sigma_1 \mid \cdots \mid c_n\sigma_n) & = & \text{size}(\Gamma \vdash \sigma_1) + \cdots + \text{size}(\Gamma \vdash \sigma_n) \\ \text{size}(\Gamma \vdash \mu\beta.\tau) & = & \text{size}(\Gamma \vdash \sigma). \end{array}$$

Ignoring the normalization process we may abstract context lattices to product lattices. Since $\text{length}(P_1 \times \cdots \times P_n) = \text{length}(P_1) + \cdots + \text{length}(P_n)$ and $\text{card}(P_1 \times \cdots \times P_n) = \text{card}(P_1) * \cdots * \text{card}(P_n)$, we have the following

6.26 Fact *Let σ be a type, then*

$$\begin{array}{rcl} \text{length}(\|\Gamma \vdash \sigma\|) & \leqslant & \text{size}(\Gamma \vdash \sigma), \\ \text{card}(\|\Gamma \vdash \sigma\|) & \leqslant & 2^{\,\text{size}(\Gamma \vdash \sigma)}. \end{array}$$

Thus for monomorphic types the length of $\|\sigma\|$ is bounded by $\text{size}(\sigma)$ while the cardinality is only bounded by $2^{\,\text{size}(\sigma)}$. The upper bounds are even tight as the example of enumeration types shows. Let $\sigma_n = c_1\varepsilon \mid \cdots \mid c_n\varepsilon$ be an enumeration of n nullary constructors, then $\text{size}(\sigma_n) = n$, $\text{length}(\|\sigma_n\|) = n$, and $\text{card}(\|\sigma_n\|) = 2^n$. For polymorphic types the situation even becomes worse, but recall the remark after Lemma 6.10.

6.5 Bibliographic notes

The representation of projections is largely motivated by the work of Kubiak *et. al.* [105]. The concept of polymorphic contexts, however, is original to our work. This chapter may be considered as putting [105] on a formal basis which is in most parts straightforward but has its subtleties (mostly due to the fact that $\cdot \otimes \cdots \otimes \cdot$ is not free).

Chapter 7

Approximating Semantics

A comment on schedules: Ok, how long will it take? For each manager involved in initial meetings add one month. For each manager who says 'data flow analysis' add another month. For each unique end-user type add one month. For each unknown software package to be employed add two months. For each unknown hardware device add two months. For each 100 miles between developer and installation add one month. For each type of communication channel add one month. If an IBM mainframe shop is involved and you are working on a non-IBM system add 6 months. If an IBM mainframe shop is involved and you are working on an IBM system add 9 months. Round up to the nearest half-year.

— Brad Sherman

This chapter introduces an effective and safe approximation of the non-standard semantics. The definition of the approximating semantics is very close to its 'prototype' the most notable exceptions being the use of independent contexts and the explicit treatment of polymorphism and recursive types.

It goes without saying that the analysis is a typed one: An expression of type σ is analysed only with respect to contexts on σ. Accordingly, a polymorphic user-defined function is only analysed at its most general type. [Recall that the most general type, commonly called the principal type, subsumes each other legal type of the function.] The infinite context transformer of `append`, for example, is merely tabulated for the contexts B, PH γ, PL γ, HT γ, H γ, T γ, and L γ, as every other demand on lists is subsumed by one of these contexts. For obvious reasons the seven contexts on list α are termed principal contexts.

The special treatment of polymorphic functions makes the analysis compatible—at least in principle—with a module concept. The finite part of a context transformer, which records the images of the principal contexts, may be stored in the interface of a module so that higher modules merely have to look up the relevant strictness information in the interface. While this works perfectly well for 'small' types such as lists having seven or trees having eleven principal contexts, the method becomes unwieldy for 'bigger' types since the size of the tables grows exponentially to the size of the underlying type. But that's the story of the generic analysis to be told in the next chapter.

Plan of the chapter Section 7.1 shows how to deal with monomorphic system-defined and polymorphic user-defined functions. The abstract variants of operations on lift and sequence types are introduced in Section 7.2 and Section 7.3 respectively. Operators dealing with contexts on data types especially recursive data types are discussed in Section 7.4. The results will be brought together in Section 7.5 which defines the approximating semantics. In Section 7.6 a bunch of examples including many standard functions on lists and trees is analysed assessing the power and the limitations of the approximating semantics. We conclude with a discussion of the run-time complexity in Section 7.7 preparing the ground for the generic analysis.

To ease the notational burden somewhat we agree upon the following

7.1 Convention *For this chapter we fix the set of assumptions* Γ *such that for each generic type variable* $\alpha \in$ **tvar** *there is a fixed but infinite number of contexts variables. Formally, let* **cvar** $:= \{ (\alpha, i) \mid \alpha \in \textbf{tvar} \wedge i \in \mathbb{N} \}$ *and set* $\Gamma := \{ (\alpha, i) : \|\alpha\| \mid (\alpha, i) \in \textbf{tvar} \}$*. Then* $\|\sigma\|$ *serves as an abbreviation for* $\|\Gamma \vdash \sigma\|$*.*

7.1 System- and user-defined functions

This and the next three sections introduce abstract variants of the operators employed in the non-standard semantics. Recall from the desiderata listed in Section 6.4 that we have to ensure that contexts are kept in normalform, that the operators are monotone wrt $\lesssim$ and that they are safe wrt their concrete counterparts.

The context transformers of system-defined functions are fixed in advance. Recall that basic types are abstracted to the two-point-domain **2**. The context transformer of the function $s : \varsigma_1? \ldots \varsigma_n? \to \varsigma$ is denoted by s^α, for instance:

$$
\begin{array}{rclrcl}
\mathtt{+}^\alpha(\mathtt{bot}) & = & \mathtt{bot!}\,\mathtt{bot!} & \mathtt{*}^\alpha(\mathtt{bot}) & = & \mathtt{bot!}\,\mathtt{bot!} \\
\mathtt{+}^\alpha(\mathtt{ide}) & = & \mathtt{ide!}\,\mathtt{ide!} & \mathtt{*}^\alpha(\mathtt{ide}) & = & \mathtt{ide!}\,\mathtt{ide?}.
\end{array}
$$

The context transformer of `+` states that the function is strict in both arguments while the multiplication is classified as being strict in the first argument only. The abstraction s^α of a system-defined function must be monotone and safe with respect to $[\![s]\!]^\sharp$, that is, for all normal contexts $\kappa \in \|\varsigma\|$,

$$(\forall \nu)\ [\![s]\!]^\sharp(\mathcal{C}[\![\kappa]\!]\nu) \sqsubseteq \mathcal{C}(s^\alpha(\kappa))\nu.$$

[Note that the assignment ν is superfluous since system-defined functions are monomorphic.]

We have described in the introduction to this chapter that the context transformer of a user-defined polymorphic function in only tabulated for the principal contexts on the result type. The analysis of a function application therefore proceeds in three steps.

1. The concrete demand κ is factored into a principal context $\hat{\kappa}$ and a substitution θ such that $\kappa = \hat{\kappa}\theta$.
2. The context transformer is applied to the principal context $\hat{\kappa}$.
3. The substitution θ is applied to the resulting context thereby evaluating all occurrences of `&` and `+`.

Let us illustrate the procedure with a simple example. The function `dup` yields a pair containing the function's argument in both components.

```
dup :: α → (α,α)
dup a = (a,a)
```

The type (α, α) gives rise to four principal contexts, namely $(\gamma_1!, \gamma_2!)$, $(\gamma_1!, \gamma_2?)$, $(\gamma_1?, \gamma_2!)$, and $(\gamma_1?, \gamma_2?)$ which specify the evaluation of both, of the first, of the second and of none component(s), respectively. The context transformer of `dup` essentially paraphrases the behaviour of $\&$ on lifts.

$$\begin{array}{lcl} \mathtt{dup}^\alpha(\gamma_1!, \gamma_2!) & = & (\gamma_1 \mathbin{\&} \gamma_2)! \\ \mathtt{dup}^\alpha(\gamma_1!, \gamma_2?) & = & (\gamma_1 + \gamma_1 \mathbin{\&} \gamma_2)! \\ \mathtt{dup}^\alpha(\gamma_1?, \gamma_2!) & = & (\gamma_1 \mathbin{\&} \gamma_2 + \gamma_2)! \\ \mathtt{dup}^\alpha(\gamma_1?, \gamma_2?) & = & (\gamma_1 + \gamma_2)? \end{array}$$

Now, assume that the concrete demand on `dup` is $(\mathrm{H}\,\gamma?, \mathrm{T}\,\gamma!)$. The demand is first factored into the principal context $(\gamma_1?, \gamma_2!)$ and the substitution $\theta = [\gamma_1/\mathrm{H}\,\gamma, \gamma_2/\mathrm{T}\,\gamma]$. Applying the substitution θ to $\mathtt{dup}^\alpha(\gamma_1?, \gamma_2!)$ thereby evaluating $\&$ and $+$ yields $(\mathrm{H}\,\gamma \mathbin{\&^\alpha} \mathrm{T}\,\gamma \sqcup^\alpha \mathrm{T}\,\gamma)! = \mathrm{T}\,\gamma!$.

In order to formalize this procedure some notation proves helpful. Let ϑ range over type substitution, then $\kappa \in \sum \vartheta.\|\sigma\vartheta\|$ means that there is a type substitution ϑ such that $\kappa \in \|\sigma\vartheta\|$. Accordingly, $\tau^\alpha : \prod \vartheta.\|\sigma_1\vartheta\| \to \|\sigma_2\vartheta\|$ asserts that for all type substitutions ϑ and for all $\kappa \in \|\sigma_1\vartheta\|$ we have $\tau^\alpha(\kappa) \in \|\sigma_2\vartheta\|$.

Formally, a principal context is a maximal element with respect to the subsumption ordering defined below.

7.2 Definition *Let $\kappa_1, \kappa_2 \in \sum \vartheta.\|\sigma\vartheta\|$ be normal contexts. Then κ_2* subsumes *κ_1 (notation: $\kappa_1 \preccurlyeq \kappa_2$) iff there is a substitution θ such that $\kappa_1 = \kappa_2\theta$.*

Let $\equiv$ denote the equivalence relation induced by $\preccurlyeq$, that is, $\kappa_1 \equiv \kappa_2 :\Longleftrightarrow \kappa_1 \preccurlyeq \kappa_2 \wedge \kappa_2 \preccurlyeq \kappa_1$. Note that two equivalent contexts differ only in the names of the context variables, that is, the substitution θ boils down to a variable renaming. It is well-known that $\preccurlyeq$ defines a pre-order on $\sum \vartheta.\|\sigma\vartheta\|$ and a partial order on $(\sum \vartheta.\|\sigma\vartheta\|)/{\equiv}$.

7.3 Definition *Let $\kappa \in \sum \vartheta.\|\sigma\vartheta\|$ be a normal context. Then κ is termed* principal *iff $[\kappa]_\equiv$ is maximal with respect to $\preccurlyeq$.*

A context $\kappa \in \sum \vartheta.\|\sigma\vartheta\|$ is principal if it is an element of $\|\sigma\|$ and if the polymorphic subcontexts are pairwise different context variables (due to the normalization some context variables may be replaced by `bot`). A context κ on lists, for example, is principal iff

$$\kappa \;=\; \mathrm{norm}[\![\mu\beta.\mathtt{[]}\ \kappa_1 \mid \gamma\ell_1 : \beta\ell_2]\!],$$

for some $\kappa_1 \in \|\varepsilon\|$ and $\ell_1, \ell_2 \in \{!, ?\}$. It is immediate that an arbitrary context $\kappa \in \sum \vartheta.\|\sigma\vartheta\|$ may be factored into a principal context $\hat{\kappa} \in \sum \vartheta.\|\sigma\vartheta\|$ and a substitution θ such that $\kappa = \hat{\kappa}\theta$. Furthermore, the factorization is unique.

The evaluation of a context is realized by the function 'eval' defined below. Note that the safety of 'eval' is an immediate consequence of the safety of $\&^\alpha$ and $\sqcup^\alpha$.

7.4 Definition and Fact *The function* eval *evaluates a polymorphic context with respect to a given substitution.*

$$\begin{array}{rcl} \mathrm{eval}[\![\mathtt{bot}]\!]\theta & = & \mathrm{bot}(\sigma) \\ \mathrm{eval}[\![\gamma]\!]\theta & = & \theta(\gamma) \\ \mathrm{eval}[\![\kappa_1 \mathbin{\&} \kappa_2]\!]\theta & = & \mathrm{eval}[\![\kappa_1]\!]\theta \mathbin{\&^\alpha} \mathrm{eval}[\![\kappa_2]\!]\theta \\ \mathrm{eval}[\![\kappa_1 + \kappa_2]\!]\theta & = & \mathrm{eval}[\![\kappa_1]\!]\theta \sqcup^\alpha \mathrm{eval}[\![\kappa_2]\!]\theta \end{array}$$

The function eval *is generalized in the obvious way to normal contexts. For all substitutions* $\theta = [\gamma_1/\kappa_1, \ldots, \gamma_n/\kappa_n]$ *and assignments* ν *let* $\hat{\nu} = \nu[\gamma_1 \mapsto \mathcal{C}[\![\kappa_1]\!]\nu, \ldots, \gamma_n \mapsto \mathcal{C}[\![\kappa_n]\!]\nu]$. *Then* eval *is monotone and for all contexts* κ *and for all substitutions* θ

$$(\forall \nu)\ \mathcal{C}[\![\kappa]\!]\hat{\nu} \sqsubseteq \mathcal{C}(\mathrm{eval}[\![\kappa]\!]\theta)\nu.$$

The correctness criterion of the non-standard semantics (cf to Definition 5.20) is weakened to the property of safety for the approximating semantics. The safety property basically ensures that a context transformer never falsely signals strictness.

7.5 Definition *Let* τ *be a projection transformer and let* $\tau^\alpha : \prod \vartheta.\|\sigma_1\vartheta\| \to \|\sigma_2\vartheta\|$ *be a context transformer, then* τ^α is safe wrt τ *iff for all type substitutions* ϑ *and for all* $\kappa \in \|\sigma_1\vartheta\|$,

$$(\forall \nu)\ \tau(\mathcal{C}[\![\kappa]\!]\nu) \sqsubseteq \mathcal{C}(\tau^\alpha(\kappa))\nu.$$

The application of a polymorphic context transformer to a context is formalized by the following definition.

7.6 Definition and Lemma *Let* $\tau^\alpha : \prod \vartheta.\|\sigma_1\vartheta\| \to \|\sigma_2\vartheta\|$ *be a context transformer, let* ϑ *be a type substitution and let* $\kappa \in \|\sigma_1\vartheta\|$ *be a context. Then* $\tau^\alpha \cdot \kappa \in \|\sigma_2\vartheta\|$ *is defined by*

$$\tau^\alpha \cdot \kappa \quad = \quad \mathrm{eval}(\tau^\alpha(\hat{\kappa}))\theta \textbf{ where } \hat{\kappa}\theta = \kappa.$$

The operation '$\cdot$' is monotonic. Let τ *be a projection transformer. If* τ^α *is safe wrt* τ, *then* $\boldsymbol{\lambda}\kappa.\tau^\alpha \cdot \kappa$ *is safe wrt* τ.

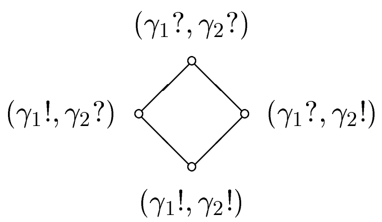
Proof Let $\theta = [\gamma_1/\kappa_1, \ldots, \gamma_n/\kappa_n]$, let $\hat{\nu} = \nu[\gamma_1 \mapsto \mathcal{C}[\![\kappa_1]\!]\nu, \ldots, \gamma_n \mapsto \mathcal{C}[\![\kappa_n]\!]\nu]$ be the assignment corresponding to θ and let $\hat{\kappa}\theta = \kappa$.

$$\begin{aligned}
\mathcal{C}(\tau^\alpha \cdot \kappa)\nu &= \mathcal{C}(\mathrm{eval}(\tau^\alpha(\hat{\kappa}))\theta)\nu && \text{def. `$\cdot$'}\\
&\sqsupseteq \mathcal{C}(\tau^\alpha(\hat{\kappa}))\hat{\nu} && \text{safety of `eval'}\\
&\sqsupseteq \tau(\mathcal{C}[\![\hat{\kappa}]\!]\hat{\nu}) && \tau^\alpha \text{ is safe wrt } \tau\\
&= \tau(\mathcal{C}[\![\kappa]\!]\nu) && \text{Substitution Lemma and } \hat{\kappa}\theta = \kappa \ \blacksquare
\end{aligned}$$

Let us identify the principal contexts for our standard set of types.

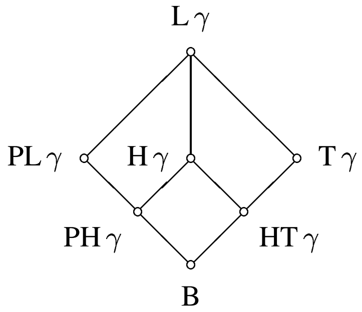
Truth values Since contexts on `bool` do not contain context variables every context is principal. This holds generally for all monomorphic types.

Polymorphic pairs The type (α_1, α_2) of polymorphic pairs gives rise to four different principal context which are displayed below.



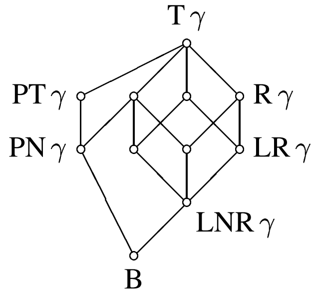
Note that this set remains the same (up to renaming) if we identify the types of the components (α, α) (cf also to the context transformer of `dup`).

Polymorphic lists In Section 6.3 we have introduced seven abbreviations for contexts on lists: B, PH γ, PL γ, HT γ, H γ, T γ, and L γ. As a matter of fact they exactly form the set of principal contexts.



As an immediate consequence the context transformer for `append`, `reverse` etc comprise seven equations. If we instantiate polymorphic lists the number of principal contexts rapidly increases: `[bool]` has 23, `[[`α`]]` has 41, and `[[bool]]` has 137 principal contexts. Even more impressing `([`α`],[`α`])` has 170 and `([bool],[bool])` has 442 principal contexts.

Polymorphic trees There are eleven principal contexts on polymorphic trees all of which are listed in Section 6.3. The Hasse diagram of the set of principal contexts is quite similar to the one for lists except that the sublattice $\mathbf{2}^2$ is replaced by $\mathbf{2}^3$ (cf to the discussion at the end of Section 6.3).



7.2 Operations on lift types

The guard operator on contexts is a direct conversion of its prototype on projections (cf to Definition 5.10).

7.7 Definition and Fact *Let $\kappa \in \|\sigma\|$ be a normal context. The context $\ell \rhd^\alpha \kappa \in \|\sigma\|$ with $\ell \in \{!, ?\}$ is defined by*

$$\begin{aligned} ! \rhd^\alpha \kappa &= \kappa \\ ? \rhd^\alpha \kappa &= \kappa \sqcup^\alpha \mathrm{abs}(\sigma). \end{aligned}$$

The guard operator $\rhd^\alpha$ is monotone and safe with respect to $\rhd$, that is, for all normal contexts $\kappa \in \|\sigma\|$ and for all lifts $\ell \in \{!, ?\}$,

$$(\forall \nu)\ [\![\ell]\!] \rhd \mathcal{C}[\![\kappa]\!]\nu \sqsubseteq \mathcal{C}(\ell \rhd^\alpha \kappa)\nu,$$

where $[\![!]\!] = \textcircled{\scriptsize{|}}$ and $[\![?]\!] = \bot$.

7.3 Operations on sequence types

Recall from Section 5.4 that the least abstraction of the projection function $\mathbf{on}_i$ which is used to access variables in the run-time environment is the projection transformer $(i\colon \cdot)$. Its counterpart on contexts is defined below.

7.8 Definition and Fact *Let $\kappa \in \|\sigma_i\|$ be a normal context. The context $(i\colon \kappa)^\alpha \in \|\sigma_1 \ldots \sigma_n\|$ is defined by*

$$(i\colon \kappa)^\alpha \quad = \quad \mathrm{norm}[\![\kappa_1 \ldots \kappa_n]\!] \textbf{ where } \kappa_j = \begin{cases} \kappa & \textbf{if } i = j \\ \mathrm{abs}(\sigma_j) & \textbf{otherwise}. \end{cases}$$

The operator $(i\colon \cdot)^\alpha$ is monotone and safe with respect to $(i\colon \cdot)$, that is, for all normal contexts $\kappa \in \|\sigma\|$,

$$(\forall \nu)\ (i\colon \mathcal{C}[\![\kappa]\!]\nu) = \mathcal{C}((i\colon \kappa)^\alpha)\nu.$$

Since empty sequences are treated separately from sequence types we need to define an abstract variant of $\langle \tau_1, \ldots, \tau_n \rangle$ for $n = 0$.

7.9 Definition and Fact *Let $\kappa \in \|\varepsilon\|$ be a normal context, then $\langle\rangle^\alpha(\kappa) \in \|\sigma\|$ is defined by*

$$\begin{aligned} \langle\rangle^\alpha(\varepsilon\mathtt{bot}) &= \mathrm{bot}(\sigma), \\ \langle\rangle^\alpha(\varepsilon\mathtt{ide}) &= \mathrm{abs}(\sigma). \end{aligned}$$

The operator $\langle\rangle^\alpha$ is monotone and safe wrt $\langle\rangle$, that is, for all normal contexts $\kappa \in \|\varepsilon\|$,

$$(\forall \nu)\ \langle\rangle(\mathcal{C}[\![\kappa]\!]\nu) = \mathcal{C}(\langle\rangle^\alpha(\kappa))\nu.$$

7.4 Operations on data types and recursive types

The following two definitions introduce variants of the operators $\cdot \downarrow i$ and $[i\colon \cdot]$.

7.10 Definition and Fact *Let $c_1\kappa_1 \mid \cdots \mid c_n\kappa_n \in \|c_1\sigma_1 \mid \cdots \mid c_n\sigma_n\|$ be a normal context. The context $(c_1\kappa_1 \mid \cdots \mid c_n\kappa_n) \downarrow^\alpha i \in \|\sigma_i\|$ is defined as follows.*

$$(c_1\kappa_1 \mid \cdots \mid c_n\kappa_n) \downarrow^\alpha i \quad = \quad \kappa_i.$$

The operator $\cdot \downarrow^\alpha i$ is monotone and safe with respect to $\cdot \downarrow i$, that is, for all normal contexts $\kappa \in \|c_1\sigma_1 \mid \cdots \mid c_n\sigma_n\|$,

$$(\forall \nu)\ \mathcal{C}[\![\kappa]\!]\nu \downarrow i = \mathcal{C}(\kappa \downarrow^\alpha i)\nu.$$

7.11 Definition and Fact *Let $\kappa \in \|\sigma_i\|$ be a normal context. The context $[i\colon \kappa]^\alpha \in \|c_1\sigma_1 \mid \cdots \mid c_n\sigma_n\|$ is defined as follows.*

$$[i\colon \kappa]^\alpha \quad = \quad c_1\kappa_1 \mid \cdots \mid c_n\kappa_n \textbf{ where } \kappa_j = \begin{cases} \kappa & \textbf{if } i = j \\ \mathrm{bot}(\sigma_j) & \textbf{otherwise} \end{cases}$$

The operator $[i\colon \cdot]^\alpha$ is monotone and safe with respect to $[i\colon \cdot]$, that is, for all normal contexts $\kappa \in \|\sigma\|$,

$$(\forall \nu)\ [i\colon \mathcal{C}[\![\kappa]\!]\nu] = \mathcal{C}([i\colon \kappa]^\alpha)\nu.$$

In functional languages like MIRANDA, HASKELL or STANDARD ML recursive types may only be introduced via data type definitions. In order to simplify the presentation of the approximating semantics it is convenient to make the same restriction.

Recall that we have confined ourselves to uniform contexts on recursive types. This fact has to be taken into account when the equations of the non-standard semantics which deal with data types are adapted for the approximating semantics. Let us illustrate the point by means of a simple recursive type, say, polymorphic lists. [In the discussion following we will use the textual representation $\mu\beta.\text{Nil}\ \varepsilon \mid \text{Cons}\ \alpha?\ \beta?$ rather than the symbolic $\mu\beta.\,\texttt{[]}\ \varepsilon \mid \alpha?:\beta?.$] A direct translation of the constructor rule in Table 5.3 specialized to the list constructor 'Cons' yields the equation

$$\mathcal{E}^\alpha[\![\text{Cons}\ e]\!]\ \phi^\alpha\ \kappa \quad = \quad \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (\kappa \downarrow^\alpha \text{Cons}).$$

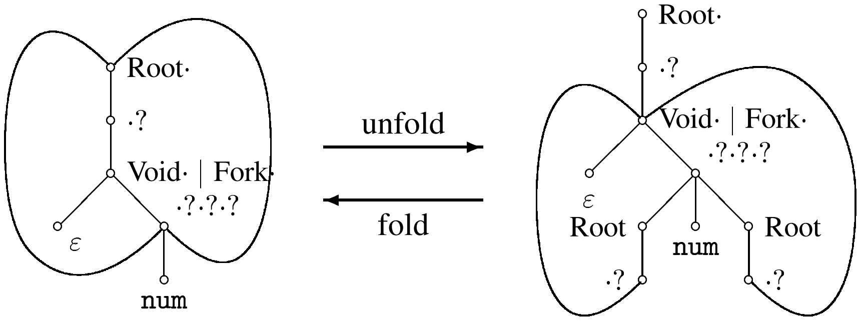
Assume that Cons e is of type $\texttt{[}\sigma\texttt{]}$ which implies that κ is an element of $\|\,\texttt{[}\sigma\texttt{]}\,\|$. Now, since the operator $\cdot \downarrow^\alpha$ Cons expects a context on a sum type we are forced to *unfold* κ prior to its application. Unfolding $\kappa = \mu\beta.\text{Nil}\ \kappa_1 \mid \text{Cons}\ \kappa_2\ell_1\ \beta\ell_2$ yields $\text{Nil}\ \kappa_1 \mid \text{Cons}\ \kappa_2\ell_1\ \kappa\ell_2$, so that the analysis continues with respect to the context $\kappa_2\ell_1\ \kappa\ell_2$.

The dual problem appears in the $\underline{\texttt{case}}$-rule which reads as follows.

$$\begin{array}{ll} & \mathcal{E}^\alpha[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ \text{Nil} \to e_1 \mid \text{Cons} \to e_2]\!]\ \phi^\alpha\ \kappa \\ = & \rho_1^\alpha \mathbin{\&^\alpha} \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ [\text{Nil}\colon \kappa_1]^\alpha \sqcup^\alpha \rho_2^\alpha \mathbin{\&^\alpha} \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ [\text{Cons}\colon \kappa_2]^\alpha \\ & \quad \textbf{where}\ \rho_1^\alpha\ \kappa_1 = \mathcal{E}^\alpha[\![e_1]\!]\ \phi^\alpha\ \kappa \\ & \qquad\qquad\ \rho_2^\alpha\ \kappa_2 = \mathcal{E}^\alpha[\![e_2]\!]\ \phi^\alpha\ \kappa \end{array}$$

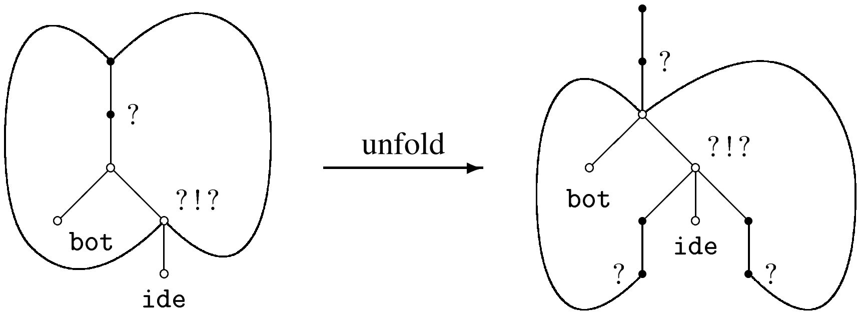
Assume now that e is of type $\texttt{[}\sigma\texttt{]}$. Since κ_1 is a context on ε and κ_2 on $\sigma?\texttt{[}\sigma\texttt{]}?$ neither $[\text{Nil}\colon \kappa_1]^\alpha$ nor $[\text{Cons}\colon \kappa_2]^\alpha$ meets the type requirement. Both contexts must be *folded* beforehand. Folding the context $\text{Nil}\ \kappa_1 \mid \text{Cons}\ \kappa_2\ell_1\ \kappa_3\ell_2$ yields $(\mu\beta.\text{Nil}\ \kappa_1 \mid \text{Cons}\ \kappa_2\ell_1\ \beta\ell_2) \sqcup^\alpha \kappa_3$. Contrary to unfolding folding generally looses information since a non-uniform context is mapped to a uniform context.

In the case of polymorphic lists unfolding corresponds to a simple substitution mapping $\mu\beta.\kappa$ to $\kappa[\beta/\mu\beta.\kappa]$. Things become more complicated if the data type definitions are *mutually* recursive. Recall the definitions of `shrub` and `node` introduced in Section 2.2 and note that the type expression 'shrub' cannot be unfolded to 'Root(node)' by means of a simple substitution. An appropriate presentation of the unfolding process is based on the graph representation of the corresponding types. The graphs for the type expressions 'shrub' and 'Root(node)' are displayed below. Note that we have collapsed every combination of sequence and lift types to a single node.



The graphs essentially comprise the term-structure of the type expressions; only μ-bound variables are represented by backarcs which point to their respective binding places. Based on this presentation unfold and fold on contexts may be explained as follows.

unfold Consider the data type definition $t\ \alpha_1 \ldots \alpha_n ::= \sigma$. The type expression stemming from its right hand side, that is, expand$[\![\sigma]\!]\ \varnothing\ \varnothing$ is used as a template for the context to produce. [Recall that a context is understood as a type expression incorporating strictness annotations.] The given context and the template are simultaneously traversed thereby copying the annotations from the given context to the template. Since the unfolded context is strictly greater than the folded context we must follow the backarcs in the given context but ignore the backarcs in the template. The operation is illustrated below. For better readability we have omitted the type constructors.



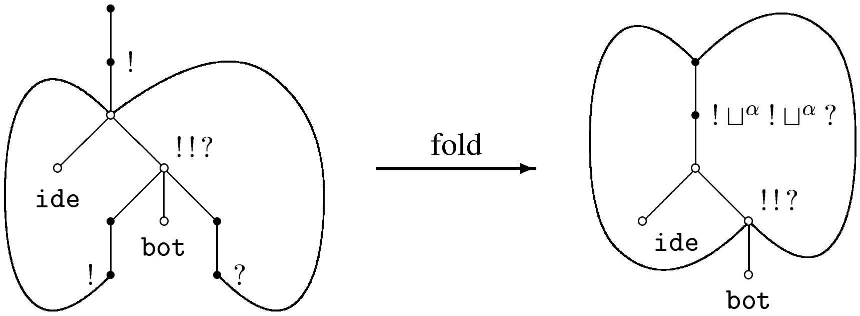
Note that the graph on the right contains three copies of the two shaded elements on the left. Hence both of these nodes are visited three times during the tree traversal.

Interpreted as rational trees[1] both graphs denote in fact the same tree which implies that their denotation is equal as well.

7.12 Fact *The operator* unfold *is monotone and for all normal contexts* $\kappa \in \|\sigma\|$,

$$(\forall \nu)\ \mathcal{C}[\![\kappa]\!]\nu = \mathcal{C}(\text{unfold}(\kappa))\nu.$$

fold The folding of contexts works with one exception completely analogous. Consider again the data type definition $t\ \alpha_1 \ldots \alpha_n ::= \sigma$. Now the type expression stemming from the left hand side, that is, expand$[\![t\ \alpha_1 \ldots \alpha_n]\!]\ \varnothing\ \varnothing$ is used as a template which is filled by a tree traversal of the given context. However, since the folded context is strictly smaller some of its nodes are visited more than once. In each of these cases the different annotations must be approximated by their least upper bound.



The three shaded subgraphs on the left are united into one subgraph on the right whose annotation equals the join of the respective operations on the left. Due to the join the folded context is usually only an approximation of the unfolded one.

7.13 Fact *The operator* fold *is monotone and for all normal contexts* $\kappa \in \|\sigma\|$,

$$(\forall \nu)\ \mathcal{C}[\![\kappa]\!]\nu \sqsubseteq \mathcal{C}(\text{fold}(\kappa))\nu.$$

[1] A rational tree is an infinite tree which contains only a finite number of different subtrees.

7.5 Semantic domains and equations

This section introduces the approximating semantics and proves it save with respect to the non-standard semantics.

Following our routine program we begin with a specification of the domains involved. The domains of the approximating semantics are shown in Table 7.1. To discriminate between

meta variable		domain	comment
κ	$\in$	$\mathbf{Val}^\alpha$	abstract values
φ^α	$\in$	$\mathbf{Fun}^\alpha$	abstract functions
ρ^α	$\in$	$\mathbf{ValEnv}^\alpha$	abstract value environments
ϕ^α	$\in$	$\mathbf{FunEnv}^\alpha$	abstract function environments

Table 7.1: Domains of the approximating semantics

the non-standard and the approximating semantics we consistently annotate the entities of the latter with the greek letter α.

Recall that an expression is interpreted relative to a given environment. Due to the use of de Bruijn indices an environment is in fact an 'ordinary' value, namely a nested pair of sequences. Accordingly the type of an environment may be expressed by an 'ordinary' type expression. Table 7.2 introduces a subcategory of **texp** which comprises legal types of environments. The domains of the approximating semantics are then defined via domain equations

production			comment
typeenv	$::=$	ε	type of empty environment
	$\mid$	$\mathbf{typeenv}\,(\mathbf{texp}_1? \ldots \mathbf{texp}_n?)$	type of environment layer

Table 7.2: Abstract syntax of an environment's type

shown in Table 7.3.

domain equation			comment
$\mathbf{Val}^\alpha$	$=$	$\lVert\mathbf{texp}\rVert$	contexts
$\mathbf{Fun}^\alpha$	$=$	$\lVert\mathbf{texp}\rVert \to \lVert\mathbf{texp}? \ldots \mathbf{texp}?\rVert$	context transformers
$\mathbf{ValEnv}^\alpha$	$=$	$\lVert\mathbf{typeenv}\rVert$	abstract value environments
$\mathbf{FunEnv}^\alpha$	$=$	$\mathbf{fun} \to \mathbf{Fun}^\alpha$	abstract function environments

Table 7.3: Domain equations of the approximating semantics

The abstraction of $\mathbf{acc}_k$ which accesses the k-th entry in the run-time environment is given by $\langle k\colon \cdot\rangle^\alpha \in \lVert\mathbf{texp}_1 \ldots \mathbf{texp}_n\rVert \to \lVert\mathbf{typeenv}\rVert$,

$$\begin{aligned}\langle 0\colon \kappa\rangle^\alpha &= (2\colon \kappa)^\alpha\\ \langle k+1\colon \kappa\rangle^\alpha &= (1\colon \langle k\colon \kappa\rangle^\alpha)^\alpha.\end{aligned}$$

It is immediate that $\langle k\colon \cdot\rangle^\alpha$ is monotone and safe with respect to $\langle k\colon \cdot\rangle$.

The equations of the approximating semantics are listed in Table 7.4. Note that the **lfp**

$\mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha$ denotes a context transformer which is safe with respect to $\mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp$ provided ϕ^α is safe with respect to $\phi^\sharp$.

$$
\begin{array}{rcl}
\mathcal{E}^\alpha[\![\mathbf{dbe}]\!] & : & \mathbf{FunEnv}^\alpha \to \mathbf{Val}^\alpha \to \mathbf{ValEnv}^\alpha \\
\mathcal{E}^\alpha[\![k.i]\!]\ \phi^\alpha\ \kappa & = & \langle k\colon (i\colon \kappa)^\alpha\rangle^\alpha \\
\mathcal{E}^\alpha[\![\texttt{freeze}\ e]\!]\ \phi^\alpha\ (\kappa\ell) & = & \ell \rhd^\alpha \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ \kappa \\
\mathcal{E}^\alpha[\![\texttt{unfreeze}\ e]\!]\ \phi^\alpha\ \kappa & = & \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (\kappa!) \\
\mathcal{E}^\alpha[\![s\ e]\!]\ \phi^\alpha\ \kappa & = & \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (s^\alpha(\kappa)) \\
\mathcal{E}^\alpha[\![c\ e]\!]\ \phi^\alpha\ \kappa & = & \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (\mathrm{unfold}(\kappa) \downarrow^\alpha c) \\
\mathcal{E}^\alpha[\![f\ e]\!]\ \phi^\alpha\ \kappa & = & \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (\phi^\alpha(f)\cdot\kappa) \\
\mathcal{E}^\alpha[\![\varepsilon]\!]\ \phi^\alpha\ \kappa & = & \langle\rangle^\alpha(\kappa) \\
\mathcal{E}^\alpha[\![e_1 \ldots e_n]\!]\ \phi^\alpha\ (\kappa_1 \ldots \kappa_n) & & \\
 & = & \mathcal{E}^\alpha[\![e_1]\!]\ \phi^\alpha\ \kappa_1 \ \&^\alpha \ldots \&^\alpha\ \mathcal{E}^\alpha[\![e_n]\!]\ \phi^\alpha\ \kappa_n \\
\mathcal{E}^\alpha[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1 \mid \cdots \mid c_n \to e_n]\!]\ \phi^\alpha\ \kappa & & \\
 & = & \rho_1^\alpha\ \&^\alpha\ \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (\mathrm{fold}[c_1\colon \kappa_1]^\alpha) \\
 & & \sqcup^\alpha \cdots \sqcup^\alpha \\
 & & \rho_n^\alpha\ \&^\alpha\ \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (\mathrm{fold}[c_n\colon \kappa_n]^\alpha) \\
 & & \textsf{where } \rho_1^\alpha\ \kappa_1 = \mathcal{E}^\alpha[\![e_1]\!]\ \phi^\alpha\ \kappa \\
 & & \qquad\quad \cdots \\
 & & \qquad\quad \rho_n^\alpha\ \kappa_n = \mathcal{E}^\alpha[\![e_n]\!]\ \phi^\alpha\ \kappa \\
\mathcal{E}^\alpha[\![\underline{\texttt{let}}\ e_1\ \underline{\texttt{in}}\ e]\!]\ \phi^\alpha\ \kappa & = & \rho^\alpha\ \&^\alpha\ \mathcal{E}^\alpha[\![e_1]\!]\ \phi^\alpha\ \kappa' \\
 & & \textsf{where } \rho^\alpha\ \kappa' = \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ \kappa \\
\mathcal{E}^\alpha[\![\underline{\texttt{rec}}\ e]\!]\ \phi^\alpha\ \kappa & = & \textsf{lfp}_{\tau_0^\alpha}(\boldsymbol{\lambda}\tau^\alpha.\boldsymbol{\lambda}\kappa.\rho^\alpha\ \&^\alpha\ \tau^\alpha(\kappa') \\
 & & \qquad \textsf{where } \rho^\alpha\ \kappa' = \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ \kappa)(\kappa) \\
 & & \textsf{where } \tau_0^\alpha = \boldsymbol{\lambda}\kappa.\mathrm{abs}(\sigma)
\end{array}
$$

$\mathcal{P}^\alpha[\![f_1 = \lambda e_1; \ldots ; f_n = \lambda e_n]\!]$ denotes an abstract function environment which is safe with respect to $\mathcal{P}^\sharp[\![f_1 = \lambda e_1; \ldots ; f_n = \lambda e_n]\!]$.

$$
\begin{array}{rcl}
\mathcal{P}^\alpha[\![\mathbf{dbp}]\!] & : & \mathbf{FunEnv}^\alpha \\
\mathcal{P}^\alpha[\![f_1 = \lambda e_1; \ldots ; f_n = \lambda e_n]\!] & & \\
 & = & \textsf{lfp}(\boldsymbol{\lambda}\phi^\alpha.\varnothing[f_1 \mapsto \boldsymbol{\lambda}\kappa.\kappa' \textsf{ where } \rho^\alpha\ \kappa' = \mathcal{E}^\alpha[\![e_1]\!]\ \phi^\alpha\ \kappa] \cdots \\
 & & \qquad\qquad [f_n \mapsto \boldsymbol{\lambda}\kappa.\kappa' \textsf{ where } \rho^\alpha\ \kappa' = \mathcal{E}^\alpha[\![e_n]\!]\ \phi^\alpha\ \kappa])
\end{array}
$$

Table 7.4: The semantic functions $\mathcal{E}^\alpha$ and $\mathcal{P}^\alpha$.

operation which occurs in both $\mathcal{E}^\alpha$ and $\mathcal{P}^\alpha$ is well-defined since the auxiliary functions employed are monotonic and the domains involved are finite. Many of the equations should be familiar from the non-standard semantics. The most striking differences may be noted in the last four equations dealing with sequences, case analysis and local definitions. The differences are mainly due to the use of independent contexts. In so far the equations listed in Table 7.4 are really simplifications of their corresponding prototypes. Consider, for example, the least abstraction of a sequence of expressions.

$$\mathcal{E}^\sharp[\![e_1 \ldots e_n]\!]\ \phi^\sharp\ \pi \ = \ \langle \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp, \ldots, \mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp\rangle(\pi)$$

If the projection π is independent, say, $\pi = \pi_1 \otimes \cdots \otimes \pi_n$ the above equation simplifies to

$$\mathcal{E}^\sharp[\![e_1 \dots e_n]\!]\ \phi^\sharp\ (\pi_1 \otimes \cdots \otimes \pi_n) \quad = \quad \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp\ \pi_1\ \&\ \cdots\ \&\ \mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp\ \pi_n,$$

which corresponds exactly to the equation of the approximating semantics. The remaining equations may be obtained in a similar way.

7.14 Definition *Let $\phi^\sharp$ be a non-standard function environment and let ϕ^α be an approximating function environment. Then ϕ^α is called* safe wrt $\phi^\sharp$ *iff $\phi^\alpha(f)$ is safe wrt $\phi^\sharp(f)$, for all $f \in$* **fun**.

7.15 Safety of the approximating semantics

1. *If ϕ^α is safe wrt $\phi^\sharp$, then $\mathcal{E}^\alpha[\![e_0]\!]\ \phi^\alpha$ is safe wrt $\mathcal{E}^\sharp[\![e_0]\!]\ \phi^\sharp$, for all $e_o \in$* **dbe**.

2. *$\mathcal{P}^\alpha[\![f_1 = \lambda e_1; \dots ; f_n = \lambda e_n]\!]$ is safe wrt $\mathcal{P}^\sharp[\![f_1 = \lambda e_1; \dots ; f_n = \lambda e_n]\!]$.*

Proof Ad 1) We proceed by structural induction on e_0. Case $e_0 = k.i$:

$$\begin{array}{rcll}
\mathcal{E}^\sharp[\![k.i]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa]\!]\nu) & = & \langle k\colon (i\colon \mathcal{C}[\![\kappa]\!]\nu)\rangle & \text{def. } \mathcal{E}^\sharp \\
& = & \mathcal{C}(\langle k\colon (i\colon \kappa)^\alpha\rangle^\alpha)\nu & \text{safety of } \langle k\colon\cdot\rangle^\alpha \text{ and } (i\colon\cdot)^\alpha \\
& = & \mathcal{C}(\mathcal{E}^\alpha[\![k.i]\!]\ \phi^\alpha\ \kappa)\nu & \text{def. } \mathcal{E}^\alpha
\end{array}$$

Case $e_0 = \texttt{freeze}\ e$, $e_0 = \texttt{unfreeze}\ e$, $e_0 = s\ e$ or $e_0 = c\ e$: Analogous using the respective safety properties and the induction hypothesis. Case $e_0 = f\ e$:

$$\begin{array}{rcll}
\mathcal{E}^\sharp[\![f\ e]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa]\!]\nu) & = & \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ (\phi^\sharp(f)(\mathcal{C}[\![\kappa]\!]\nu)) & \text{def. } \mathcal{E}^\sharp \\
& \sqsubseteq & \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ (\mathcal{C}(\phi^\alpha(f)\cdot\kappa)\nu) & [\phi^\alpha \text{ is safe wrt } \phi^\sharp\text{; safety of `}\cdot\text{'} \\
& \sqsubseteq & \mathcal{C}(\mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (\phi^\alpha(f)\cdot\kappa))\nu & \text{ind. hyp.} \\
& = & \mathcal{C}(\mathcal{E}^\alpha[\![f\ e]\!]\ \phi^\alpha\ \kappa)\nu) & \text{def. } \mathcal{E}^\alpha
\end{array}$$

Case $e_0 = \varepsilon$: Immediate by the safety of $\langle\rangle^\alpha$. Case $e_0 = e_1 \dots e_n$: First note that for $n > 0$,

$$\langle \tau_1, \dots, \tau_n\rangle(\pi_1 \otimes \cdots \otimes \pi_n) \quad = \quad \tau_1(\pi_1)\ \&\ \cdots\ \&\ \tau_n(\pi_n). \tag{7.1}$$

Using this property we have,

$$\begin{array}{rll}
& \mathcal{E}^\sharp[\![e_1 \dots e_n]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa_1 \dots \kappa_n]\!]\nu) & \\
= & \langle \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp, \dots, \mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp\rangle(\mathcal{C}[\![\kappa_1 \dots \kappa_n]\!]\nu) & \text{def. } \mathcal{E}^\sharp \\
= & \langle \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp, \dots, \mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp\rangle(\mathcal{C}[\![\kappa_1]\!]\nu \otimes \cdots \otimes \mathcal{C}[\![\kappa_n]\!]\nu) & \text{def. } \mathcal{C} \\
= & \mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa_1]\!]\nu)\ \&\ \cdots\ \&\ \mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa_n]\!]\nu) & \text{by (7.1)} \\
\sqsubseteq & \mathcal{C}(\mathcal{E}^\alpha[\![e_1]\!]\ \phi^\alpha\ \kappa_1)\nu\ \&\ \cdots\ \&\ \mathcal{C}(\mathcal{E}^\alpha[\![e_n]\!]\ \phi^\alpha\ \kappa_n)\nu & \text{ind. hyp.} \\
\sqsubseteq & \mathcal{C}(\mathcal{E}^\alpha[\![e_1]\!]\ \phi^\alpha\ \kappa_1\ \&^\alpha\ \cdots\ \&^\alpha\ \mathcal{E}^\alpha[\![e_n]\!]\ \phi^\alpha\ \kappa_n)\nu & \text{safety of } \&^\alpha \\
= & \mathcal{C}(\mathcal{E}^\alpha[\![e_1 \dots e_n]\!]\ \phi^\alpha\ (\kappa_1 \dots \kappa_n))\nu & \text{def. } \mathcal{E}^\alpha
\end{array}$$

Case $e_0 = \underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1 \mid \cdots \mid c_n \to e_n$: Let $\rho_i^\alpha\, \kappa_i = \mathcal{E}^\alpha[\![e_i]\!]\ \phi^\alpha\ \kappa$ and let $\tau_i(\pi) = \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ [c_i\colon \pi]$, then

$$\begin{array}{cll}
& \mathcal{E}^\sharp[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1 \mid \cdots \mid c_n \to e_n]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa]\!]\nu) & \\
= & \langle \mathbf{id}, \tau_1\rangle(\mathcal{E}^\sharp[\![e_1]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa]\!]\nu)) \sqcup \cdots \sqcup \langle \mathbf{id}, \tau_n\rangle(\mathcal{E}^\sharp[\![e_n]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa]\!]\nu)) & \text{def. } \mathcal{E}^\sharp \\
\sqsubseteq & \langle \mathbf{id}, \tau_1\rangle(\mathcal{C}[\![\rho_1^\alpha\, \kappa_1]\!]\nu) \sqcup \cdots \sqcup \langle \mathbf{id}, \tau_n\rangle(\mathcal{C}[\![\rho_n^\alpha\, \kappa_n]\!]\nu) & \text{ind. hyp.} \\
= & \mathcal{C}[\![\rho_1^\alpha]\!]\nu \ \&\ \tau_1(\mathcal{C}[\![\kappa_1]\!]\nu) \sqcup \cdots \sqcup \mathcal{C}[\![\rho_n^\alpha]\!]\nu \ \&\ \tau_n(\mathcal{C}[\![\kappa_n]\!]\nu) & \text{def. } \mathcal{C} \text{ and (7.1)} \\
\sqsubseteq & \mathcal{C}[\![\rho_1^\alpha]\!]\nu \ \&\ \mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ (\mathcal{C}(\text{fold}[c_1\colon \kappa_1]^\alpha)\nu) \sqcup \cdots \sqcup __ & [\text{safety of 'fold' and } [c\colon \cdot]^\alpha \\
\sqsubseteq & \mathcal{C}[\![\rho_1^\alpha]\!]\nu \ \&\ \mathcal{C}(\mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (\text{fold}[c_1\colon \kappa_1]^\alpha))\nu \sqcup \cdots \sqcup __ & \text{ind. hyp.} \\
\sqsubseteq & \mathcal{C}(\rho_1^\alpha \ \&^\alpha\ \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ (\text{fold}[c_1\colon \kappa_1]^\alpha) \sqcup^\alpha \cdots \sqcup^\alpha __)\nu & \text{safety of } \&^\alpha \text{ and } \sqcup^\alpha \\
= & \mathcal{C}(\mathcal{E}^\alpha[\![\underline{\texttt{case}}\ e\ \underline{\texttt{of}}\ c_1 \to e_1 \mid \cdots \mid c_n \to e_n]\!]\ \phi^\alpha\ \kappa)\nu & \text{def. } \mathcal{E}^\alpha
\end{array}$$

Case $e_0 = \underline{\texttt{let}}\ e_1\ \underline{\texttt{in}}\ e$: Analogous. Case $e_0 = \underline{\texttt{rec}}\ e$: We show the proposition by Fixpoint Induction. Note that the predicate

$$P(\tau, \tau^\alpha) \iff (\forall\kappa)(\forall\nu)\ \tau(\mathcal{C}[\![\kappa]\!]\nu) \sqsubseteq \mathcal{C}(\tau^\alpha(\kappa))\nu$$

is admissible. Induction basis:

$$\begin{array}{rcll}
\tau_0(\mathcal{C}[\![\kappa]\!]\nu) & = & (\boldsymbol{\lambda} v.(\mathbf{up}\,\bot, \ldots, \mathbf{up}\,\bot))^\sharp(\mathcal{C}[\![\kappa]\!]\nu) & \text{def. } \tau_0 \\
& \sqsubseteq & \mathsf{1}_{[\![\sigma]\!]} & \\
& = & \mathcal{C}(\text{abs}(\sigma))\nu & \text{Fact 6.16} \\
& = & \mathcal{C}(\tau_0^\alpha(\kappa))\nu & \text{def. } \tau_0^\alpha
\end{array}$$

Induction step: Assume that $P(\tau, \tau^\alpha)$ holds and let $\rho^\alpha\, \kappa' = \mathcal{E}^\alpha[\![e]\!]\ \phi^\alpha\ \kappa$.

$$\begin{array}{rcll}
\langle \mathbf{id}, \tau\rangle(\mathcal{E}^\sharp[\![e]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa]\!]\nu)) & \sqsubseteq & \langle \mathbf{id}, \tau\rangle(\mathcal{C}[\![\rho^\alpha\, \kappa']\!]\nu) & \text{outer ind. hyp.} \\
& = & \mathcal{C}[\![\rho^\alpha]\!]\nu \ \&\ \tau(\mathcal{C}[\![\kappa']\!]\nu) & \text{def. } \mathcal{C} \text{ and (7.1)} \\
& \sqsubseteq & \mathcal{C}[\![\rho^\alpha]\!]\nu \ \&\ \mathcal{C}(\tau^\alpha(\kappa'))\nu & \text{inner ind. hyp.} \\
& \sqsubseteq & \mathcal{C}(\rho^\alpha \ \&^\alpha\ \tau^\alpha(\kappa'))\nu & \text{safety of } \&^\alpha
\end{array}$$

Ad 2) We show the proposition by Fixpoint Induction. Note that the predicate

$$P(\phi^\alpha, \phi^\sharp) \iff \phi^\alpha \text{ is safe wrt } \phi^\sharp$$

is admissible. Induction basis: Trivial since $\phi^\sharp(f)(\pi) = \perp\!\!\!\perp$. Induction step: Assume that $P(\tau, \tau^\alpha)$ holds and let $\rho^\alpha\, \kappa' = \mathcal{E}^\alpha[\![e_i]\!]\ \phi^\alpha\ \kappa$. Note that $\pi \sqsubseteq \mathcal{C}[\![\kappa_1 \ldots \kappa_n]\!]\nu$ implies $\pi \uparrow i \sqsubseteq \mathcal{C}[\![\kappa_i]\!]\nu$. Hence we have $\mathcal{E}^\sharp[\![e_i]\!]\ \phi^\sharp\ (\mathcal{C}[\![\kappa]\!]\nu) \uparrow 2 \sqsubseteq \mathcal{C}[\![\kappa']\!]\nu$ by Part 1. ■

7.6 Examples

Context-based strictness analysis on the test bench: In the subsequent sections we will analyse numerous functions assessing the power and the limitations of the analysis presented above. The functions are working on a variety of types such as lists, trees, propositional formulae etc. Many functions belong to the folklore of functional programming. However, some programs such as bottom-up mergesort are probably less known. In these cases it is advisable to consult beforehand Appendix B which lists and explains the definition of every function mentioned in this dissertation.

Note that the context transformers have been inferred using a implementation of the analysis in Miranda.[2] The analyser is essentially a transcription of its mathematical definition

[2] The program owes much to an earlier implementation by Michael Kettler, see [102].

presented in this and in the previous chapter.

7.6.1 Simple functions

`if` Since the definition of the conditional function is quite short we will work out its analysis in detail.

$$\begin{array}{l}\texttt{if :: bool? } \alpha\texttt{? } \alpha\texttt{? } \to \alpha \\ \texttt{if x y z = \underline{case} unfreeze x \underline{of} True } \to \texttt{ unfreeze y |} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{False } \to \texttt{ unfreeze z}\end{array}$$

The result type of `if` is α. As a function is only analyzed at its most general type the only context we have to consider is in fact the context variable γ. The function environment ϕ^α plays no rôle because `if` does not rest upon any auxiliary function definitions. The demand γ is first propagated to the branches of the <u>`case`</u>-expression.

$$\begin{array}{lclcl}\mathcal{E}^\alpha[\![\texttt{unfreeze y}]\!]\, \phi^\alpha\, \gamma &=& \mathcal{E}^\alpha[\![\texttt{y}]\!]\, \phi^\alpha\, (\gamma!) &=& (xyz\colon \texttt{bot}?\, \gamma!\, \texttt{bot}?, \varepsilon\colon \texttt{ide}) \\ \mathcal{E}^\alpha[\![\texttt{unfreeze z}]\!]\, \phi^\alpha\, \gamma &=& \mathcal{E}^\alpha[\![\texttt{z}]\!]\, \phi^\alpha\, (\gamma!) &=& (xyz\colon \texttt{bot}?\, \texttt{bot}?\, \gamma!, \varepsilon\colon \texttt{ide})\end{array}$$

Note that the components of the resulting contexts have been annotated with the respective patterns in order to improve readability. The expression `unfreeze x` is now analysed with respect to the contexts $[\text{True}\colon \texttt{ide}]^\alpha = \text{T}$ and $[\text{False}\colon \texttt{ide}]^\alpha = \text{F}$.

$$\begin{array}{lcl}\mathcal{E}^\alpha[\![\texttt{unfreeze x}]\!]\, \phi^\alpha\, \text{T} &=& (xyz\colon \text{T}!\, \texttt{bot}?\, \texttt{bot}?) \\ \mathcal{E}^\alpha[\![\texttt{unfreeze x}]\!]\, \phi^\alpha\, \text{F} &=& (xyz\colon \text{F}!\, \texttt{bot}?\, \texttt{bot}?)\end{array}$$

It remains to combine the intermediate results according to the <u>`case`</u>-rule.

$$\begin{array}{ll}\multicolumn{2}{l}{\mathcal{E}^\alpha[\![\underline{\texttt{case}}\texttt{ unfreeze x }\ldots]\!]\, \phi^\alpha\, \gamma} \\ = & (xyz\colon \texttt{bot}?\, \gamma!\, \texttt{bot}?)\ \&^\alpha\ (xyz\colon \text{T}!\, \texttt{bot}?\, \texttt{bot}?) \sqcup^\alpha \\ & (xyz\colon \texttt{bot}?\, \texttt{bot}?\, \gamma!)\ \&^\alpha\ (xyz\colon \text{F}!\, \texttt{bot}?\, \texttt{bot}?) \\ = & (xyz\colon \text{T}!\, \gamma!\, \texttt{bot}?) \sqcup^\alpha (xyz\colon \text{F}!\, \texttt{bot}?\, \gamma!) \\ = & (xyz\colon \text{I}!\, \gamma?\, \gamma?)\end{array}$$

Note that the joint strictness of the second and the third argument is lost only in the last step as $\sqcup^\alpha$ involves an approximation. The context transformer of `if` reads as follows.

$$\texttt{if}^\alpha(\gamma) \quad = \quad \text{I}!\, \gamma?\, \gamma?$$

Hence `if` is lift strict in the first and not lift strict in the second and third argument. Since `if` is a function which is used quite often it is not wise to base the analysis of any dependent function on the rather weak context transformer above. We will use

$$\texttt{if}^\alpha(\gamma) \quad = \quad \text{T}!\, \gamma!\, \texttt{bot}? \sqcup^\alpha \text{F}!\, \texttt{bot}?\, \gamma!$$

instead treating `if` as a mere abbreviation for the <u>`case`</u>-expression on the right hand side of its definition.

`sor` We have used ‘sequential or’ as a running example in Chapter 4 and Chapter 5. Its least abstraction is tabulated on Page 69. The context transformer of `sor` amounts to

$$\begin{array}{lcl}\texttt{sor}^{\alpha}(\mathrm{B}) & = & \mathrm{B!}\ \mathrm{B!}\\ \texttt{sor}^{\alpha}(\mathrm{F}) & = & \mathrm{F!}\ \mathrm{F!}\\ \texttt{sor}^{\alpha}(\mathrm{T}) & = & \mathrm{I!}\ \mathrm{T?}\\ \texttt{sor}^{\alpha}(\mathrm{I}) & = & \mathrm{I!}\ \mathrm{I?}.\end{array}$$

Irrespective of the demand ‘sequential or’ is lift strict in the first argument and not lift strict in the second argument. If only `False` is acceptable as a result, then both arguments are definitely required and must equal `False` (second equation). Note that the third and the fourth equation involve approximations due to the use of independent contexts.

`const` The function `const` takes two arguments and returns the first one.

```
const :: α? β? → α
const a b = unfreeze a
```

According to Definition 4.20 `const` is lift strict in its first argument and ignores its second argument. These properties are reflected in its context transformer.

$$\texttt{const}^{\alpha}(\gamma) \;=\; \gamma!\ \texttt{bot}?$$

It is interesting to note that some strictness properties may already be read from the principal type of a function. If the argument type contains a type variable which does not appear in the result type, then the function obviously ignores the respective argument.

`diverge` To a very limited extent strictness analysis may also detect non-termination of a function. The simplest non-terminating function is given by the following definition.

```
diverge :: α? → β
diverge a = diverge a
```

The function `diverge` denotes the bottom function. Due to the simplicity of the definition the approximating semantics computes, in fact, the least abstraction of `diverge`.

$$\texttt{diverge}^{\alpha}(\gamma) \;=\; \texttt{bot!}$$

Again, the behaviour may be inferred from the principal type. As the result type consists of a singleton type variable `diverge` obviously diverges (whence its name).

7.6.2 Functions on pairs

The context transformers for `fst` and `snd` nicely reflect the operational behaviour of the projection functions.

$$\begin{array}{lcl}\texttt{fst}^{\alpha}(\gamma) & = & (\gamma!,\texttt{bot}?)!\\ \texttt{snd}^{\alpha}(\gamma) & = & (\texttt{bot}?,\gamma!)!\end{array}$$

The function `fst` is lift strict and strict in the first component while the second component of the pair is ignored. An analogous result holds for `snd`. The transformer of the function `dup` was already given in Section 7.1.

7.6.3 Functions on lists

append The function append concatenates its argument lists. The definition of append is repeated below.

```
append :: [α]? [α]? → [α]
append x y = case unfreeze x of
                 [] → unfreeze y |
                 a:w → a:freeze (append w y)
```

Since append is recursively defined its analysis involves a fixpoint iteration. Being the first interesting recursive function we will work out its analysis in detail. The iteration starts with the least context transformer $\texttt{append}^{\alpha}(\kappa) = \mathrm{B!\ B!}$. Assume that the demand on append is $\mathrm{PH}\,\gamma = \mu\beta.\texttt{[] bot} \mid \gamma!:\beta?$. The demand is first propagated to the branches of the case-expression.

$$\mathcal{E}^{\alpha}[\![\texttt{unfreeze y}]\!]\,\phi^{\alpha}\,(\mathrm{PH}\,\gamma) \;=\; (xy\colon \mathrm{B?\,PH}\,\gamma!, \varepsilon\colon \texttt{ide})$$

The analysis of the recursive branch uses the initial context transformer of append.

$$\begin{aligned}
&\mathcal{E}^{\alpha}[\![\texttt{a:freeze (append w y)}]\!]\,\phi^{\alpha}\,(\mathrm{PH}\,\gamma)\\
=\;& \mathcal{E}^{\alpha}[\![\texttt{a (freeze (append w y))}]\!]\,\phi^{\alpha}\,(\gamma!\,\mathrm{PH}\,\gamma?)\\
=\;& \mathcal{E}^{\alpha}[\![\texttt{a}]\!]\,\phi^{\alpha}\,(\gamma!)\ \&^{\alpha}\ \mathcal{E}^{\alpha}[\![\texttt{freeze (append w y)}]\!]\,\phi^{\alpha}\,(\mathrm{PH}\,\gamma?)\\
=\;& \mathcal{E}^{\alpha}[\![\texttt{a}]\!]\,\phi^{\alpha}\,(\gamma!)\ \&^{\alpha}\ (?\rhd^{\alpha}\mathcal{E}^{\alpha}[\![\texttt{append w y}]\!]\,\phi^{\alpha}\,(\mathrm{PH}\,\gamma))\\
=\;& \mathcal{E}^{\alpha}[\![\texttt{a}]\!]\,\phi^{\alpha}\,(\gamma!)\ \&^{\alpha}\ (?\rhd^{\alpha}\mathcal{E}^{\alpha}[\![\texttt{w y}]\!]\,\phi^{\alpha}\,(\mathrm{B!\ B!}))\\
=\;& (xy\colon \mathrm{B?\,B?}, aw\colon \gamma!\,\mathrm{B?})\ \&^{\alpha}\ (xy\colon \mathrm{B?\,B?}, aw\colon \texttt{bot}?\,\mathrm{B?})\\
=\;& (xy\colon \mathrm{B?\,B?}, aw\colon \gamma!\,\mathrm{B?})
\end{aligned}$$

The discriminator unfreeze x is now analysed with respect to $\mathrm{fold}[\mathrm{Nil}\colon \texttt{ide}]^{\alpha} = \mathrm{HT}\,\texttt{bot}$ and $\mathrm{fold}[\mathrm{Cons}\colon \gamma!\,\mathrm{B?}]^{\alpha} = \mathrm{PH}\,\gamma$.

$$\begin{aligned}
\mathcal{E}^{\alpha}[\![\texttt{unfreeze x}]\!]\,\phi^{\alpha}\,(\mathrm{HT}\,\texttt{bot}) &= (xy\colon \mathrm{HT}\,\texttt{bot}!\,\mathrm{B?})\\
\mathcal{E}^{\alpha}[\![\texttt{unfreeze x}]\!]\,\phi^{\alpha}\,(\mathrm{PH}\,\gamma) &= (xy\colon \mathrm{PH}\,\gamma!\,\mathrm{B?})
\end{aligned}$$

Combining the intermediate results we have

$$\begin{aligned}
&\mathcal{E}^{\alpha}[\![\underline{\texttt{case}}\texttt{ unfreeze x }\ldots]\!]\,\phi^{\alpha}\,(\mathrm{PH}\,\gamma)\\
=\;& (xy\colon \mathrm{B?\,PH}\,\gamma!)\ \&^{\alpha}\ (xy\colon \mathrm{HT}\,\texttt{bot}!\,\mathrm{B?}) \sqcup^{\alpha} (xy\colon \mathrm{B?\,B?})\ \&^{\alpha}\ (xy\colon \mathrm{PH}\,\gamma!\,\mathrm{B?})\\
=\;& (xy\colon \mathrm{HT}\,\texttt{bot}!\,\mathrm{PH}\,\gamma!) \sqcup^{\alpha} (xy\colon \mathrm{PH}\,\gamma!\,\mathrm{B?})\\
=\;& (xy\colon \mathrm{H}\,\gamma!\,\mathrm{PH}\,\gamma?).
\end{aligned}$$

Hence, the first round yields $\texttt{append}^{\alpha}(\mathrm{PH}\,\gamma) = \mathrm{H}\,\gamma!\,\mathrm{PH}\,\gamma?$. The remaining images of the context transformer are calculated in a similar fashion.

The analysis is now repeated on basis of the context transformer inferred in the first round. We confine ourselves to the second branch of the case since only this branch contains a recursive call to append.

$$\begin{aligned}
&\mathcal{E}^{\alpha}[\![\texttt{a:freeze (append w y)}]\!]\,\phi^{\alpha}\,(\mathrm{PH}\,\gamma)\\
=\;& \mathcal{E}^{\alpha}[\![\texttt{a}]\!]\,\phi^{\alpha}\,(\gamma!)\ \&^{\alpha}\ (?\rhd^{\alpha}\mathcal{E}^{\alpha}[\![\texttt{append w y}]\!]\,\phi^{\alpha}\,(\mathrm{PH}\,\gamma))\\
=\;& \mathcal{E}^{\alpha}[\![\texttt{a}]\!]\,\phi^{\alpha}\,(\gamma!)\ \&^{\alpha}\ (?\rhd^{\alpha}\mathcal{E}^{\alpha}[\![\texttt{w y}]\!]\,\phi^{\alpha}\,(\mathrm{H}\,\gamma!\,\mathrm{PH}\,\gamma?))\\
=\;& (xy\colon \mathrm{B?\,B?}, aw\colon \gamma!\,\mathrm{B?})\ \&^{\alpha}\ (xy\colon \mathrm{B?\,PH}\,\gamma?, aw\colon \texttt{bot}?\,\mathrm{H}\,\gamma?)\\
=\;& (xy\colon \mathrm{B?\,PH}\,\gamma?, aw\colon \gamma!\,\mathrm{H}\,\gamma?)
\end{aligned}$$

Analysing `unfreeze x` with respect to $\text{fold}[\text{Cons}\colon \gamma!\,\text{H}\,\gamma?]^\alpha = \text{H}\,\gamma$ yields

$$\mathcal{E}^\alpha[\![\texttt{unfreeze x}]\!]\;\phi^\alpha\;(\text{H}\,\gamma) \quad = \quad (xy\colon \text{H}\,\gamma!\,\text{B}?).$$

Putting the intermediate results together we get

$$\begin{aligned}
&\mathcal{E}^\alpha[\![\underline{\texttt{case}}\ \texttt{unfreeze x}\ \ldots]\!]\;\phi^\alpha\;(\text{PH}\,\gamma)\\
=\quad& (xy\colon \text{B}?\,\text{PH}\,\gamma!)\ \&^\alpha\ (xy\colon \text{HT}\,\texttt{bot}!\,\text{B}?) \sqcup^\alpha (xy\colon \text{B}?\,\text{PH}\,\gamma?)\ \&^\alpha\ (xy\colon \text{H}\,\gamma!\,\text{B}?)\\
=\quad& (xy\colon \text{HT}\,\texttt{bot}!\,\text{PH}\,\gamma!) \sqcup^\alpha (xy\colon \text{H}\,\gamma!\,\text{PH}\,\gamma?)\\
=\quad& (xy\colon \text{H}\,\gamma!\,\text{PH}\,\gamma?).
\end{aligned}$$

Alas, we have found the least fixpoint. The complete context transformer for `append` is tabulated below.

$$\begin{aligned}
\texttt{append}^\alpha(\text{B}) &= \text{B}!\ \text{B}!\\
\texttt{append}^\alpha(\text{PH}\,\gamma) &= \text{H}\,\gamma!\ \text{PH}\,\gamma?\\
\texttt{append}^\alpha(\text{PL}\,\gamma) &= \text{L}\,\gamma!\ \text{PL}\,\gamma?\\
\texttt{append}^\alpha(\text{HT}\,\gamma) &= \text{HT}\,\gamma!\ \text{HT}\,\gamma!\\
\texttt{append}^\alpha(\text{H}\,\gamma) &= \text{H}\,\gamma!\ \text{H}\,\gamma?\\
\texttt{append}^\alpha(\text{T}\,\gamma) &= \text{T}\,\gamma!\ \text{T}\,\gamma!\\
\texttt{append}^\alpha(\text{L}\,\gamma) &= \text{L}\,\gamma!\ \text{L}\,\gamma?
\end{aligned}$$

Hence `append` is lift strict in its first argument and not lift strict in its second. Note that list concatenation becomes strict in the second argument if demand is tail strict. This clearly reflects the operational behaviour of `append`. If the spine of the concatenated list is evaluated, then the second argument being a part of it is eventually accessed. Furthermore note that the demand on `append` is literately propagated to the second argument. Again this is due to the fact that the second argument is a sublist of the result.

`reverse` We have already encountered `reverse` in the introduction to this dissertation. The naïve and the linear variant of `reverse` given in Section B.3 possess the same context transformer.

$$\begin{aligned}
\texttt{reverse}^\alpha(\text{B}) &= \text{B}!\\
\texttt{reverse}^\alpha(\text{PH}\,\gamma) &= \text{T}\,\gamma!\\
\texttt{reverse}^\alpha(\text{PL}\,\gamma) &= \text{T}\,\gamma!\\
\texttt{reverse}^\alpha(\text{HT}\,\gamma) &= \text{HT}\,\gamma!\\
\texttt{reverse}^\alpha(\text{H}\,\gamma) &= \text{T}\,\gamma!\\
\texttt{reverse}^\alpha(\text{T}\,\gamma) &= \text{T}\,\gamma!\\
\texttt{reverse}^\alpha(\text{L}\,\gamma) &= \text{T}\,\gamma!
\end{aligned}$$

It is interesting that `reverse` is lift and tail strict irrespective of the actual demand. This behaviour is easily explained: to construct the first cell of the mirrored list `reverse` must traverse the complete argument list. Head strictness is not preserved, since the first element of the result is the last element of the argument list.

`head` The function `head` extracts the first element of a non-empty list, it is undefined if the list is empty. This behaviour is reflected in the context transformer.

$$\text{head}^\alpha(\gamma) \quad = \quad \text{PH}\,\gamma!$$

The context $\text{PH}\,\gamma!$ specifies that `head` is lift strict and head strict and that the empty list is not acceptable as an argument. Note that the context transformer for `head` does not represent the least abstraction since $\text{head}^\alpha(\gamma) = \mu\beta.\texttt{[] bot} \mid \gamma!\texttt{:bot}?$ is smaller and still safe. Albeit the latter context is not a proper context in the sense of Definition 6.11.

`sum` The function `sum` is the archetype of a function being both head and tail strict.

$$\begin{aligned} \texttt{sum}^{\alpha}(\texttt{bot}) &= \text{B!} \\ \texttt{sum}^{\alpha}(\texttt{ide}) &= \text{HT}\,\texttt{ide}! \end{aligned}$$

Hence the argument list may be completely evaluated in advance.

`length` The length function is the archetype of a tail strict function.

$$\begin{aligned} \texttt{length}^{\alpha}(\texttt{bot}) &= \text{B!} \\ \texttt{length}^{\alpha}(\texttt{ide}) &= \text{T}\,\texttt{bot}! \end{aligned}$$

Hence it is safe to evaluate the spine of the argument list prior to the call of `length`. Since `length` ignores the elements of the list, the code generator may produce dummy code for them. [Of course, the last optimization only makes sense if the list is created anew which is not very likely.]

`and` The logical conjunction of a list of Boolean values which is head strict completes the presentation of archetypes.

$$\begin{aligned} \texttt{and}^{\alpha}(\text{B}) &= \text{B!} \\ \texttt{and}^{\alpha}(\text{T}) &= \text{HT T!} \\ \texttt{and}^{\alpha}(\text{F}) &= \text{PH I!} \\ \texttt{and}^{\alpha}(\text{I}) &= \text{H I!} \end{aligned}$$

The behaviour of `sum`, `length` and `and` is fruitfully explained in a more general setting. It is well-known that these functions (and many others) may be defined in terms of the higher-order function `foldr`.

```
foldr (⊕) e x = case x of [] → e |
                          a:w → a ⊕ foldr (⊕) e w
sum = foldr (+) 0
length = let a ⊕ n = 1 + n in foldr (⊕) 0
and = foldr (&) True
```

The function `foldr` takes three arguments: a binary operator $\oplus$, an element e and a list `x`. Its effect is to replace the empty list in `x` by e and the '`:`' constructors in `x` by $\oplus$.

$$\begin{array}{c} a_1 : (a_2 : \cdots : (a_n : \texttt{[]}) \cdots) \\ \Big\downarrow \; \texttt{foldr}\ (\oplus)\ e \\ a_1 \oplus (a_2 \oplus \cdots \oplus (a_n \oplus e) \cdots) \end{array}$$

It is immediate that the strictness of $\oplus$ determines the strictness of `foldr` $(\oplus)$ e: If $\oplus$ is lift strict in the first argument, then `foldr` $(\oplus)$ e is head strict. Accordingly, lift strictness in the second argument implies tail strictness. The individual properties of `sum`, `length` and `and` follow easily from these considerations.

`member` The function `member` checks whether its second argument is an element of the list passed as the first argument. [Note that `member` has the type `[num] num` $\rightarrow$ `bool` instead of `[`α`]` $\alpha \rightarrow$ `bool` since the test for equality used in the definition of `member` is restricted to numerical values.]

$$\begin{array}{lcl}
\texttt{member}^{\alpha}(\mathrm{B}) & = & \mathrm{B}!\ \texttt{bot}! \\
\texttt{member}^{\alpha}(\mathrm{T}) & = & \mathrm{PH}\,\texttt{ide}!\ \texttt{ide}! \\
\texttt{member}^{\alpha}(\mathrm{F}) & = & \mathrm{HT}\,\texttt{ide}!\ \texttt{ide}? \\
\texttt{member}^{\alpha}(\mathrm{I}) & = & \mathrm{H}\,\texttt{ide}!\ \texttt{ide}?
\end{array}$$

To our surprise the second argument, namely the element being searched in the list, is classified as probably not needed! On may suspect this to be a bad approximation of the real operational behaviour, but this is not the case. If the list is empty, we have `member [] ` $\perp \Rightarrow$ `False`, hence member is indeed not lift strict in the second argument.

`concat` The function `concat` concatenates a list of lists into a single list. Its context transformer gives a nice example for nested polymorphic contexts.

$$\begin{array}{lcl}
\texttt{concat}^{\alpha}(\mathrm{B}) & = & \mathrm{B}! \\
\texttt{concat}^{\alpha}(\mathrm{PH}\,\gamma) & = & \mathrm{PH}\,(\mathrm{H}\,\gamma)! \\
\texttt{concat}^{\alpha}(\mathrm{PL}\,\gamma) & = & \mathrm{PH}\,(\mathrm{L}\,\gamma)! \\
\texttt{concat}^{\alpha}(\mathrm{HT}\,\gamma) & = & \mathrm{HT}\,(\mathrm{HT}\,\gamma)! \\
\texttt{concat}^{\alpha}(\mathrm{H}\,\gamma) & = & \mathrm{H}\,(\mathrm{H}\,\gamma)! \\
\texttt{concat}^{\alpha}(\mathrm{T}\,\gamma) & = & \mathrm{HT}\,(\mathrm{T}\,\gamma)! \\
\texttt{concat}^{\alpha}(\mathrm{L}\,\gamma) & = & \mathrm{H}\,(\mathrm{L}\,\gamma)!
\end{array}$$

We see that `concat` is strict and head strict. If the demand on the result is tail strict then the spine of the outer list is required. The embedded contexts correspond exactly to the demands on the first argument of `append`.

`take` The function `take` computes an initial segment of a given length: `take 7` x, for example, returns the first seven elements of x.

$$\begin{array}{lcl}
\texttt{take}^{\alpha}(\mathrm{B}) & = & \texttt{bot}!\ \mathrm{B}! \\
\texttt{take}^{\alpha}(\mathrm{PH}\,\gamma) & = & \texttt{ide}!\ \mathrm{PH}\,\gamma! \\
\texttt{take}^{\alpha}(\mathrm{PL}\,\gamma) & = & \texttt{ide}!\ \mathrm{PL}\,\gamma! \\
\texttt{take}^{\alpha}(\mathrm{HT}\,\gamma) & = & \texttt{ide}!\ \mathrm{H}\,\gamma? \\
\texttt{take}^{\alpha}(\mathrm{H}\,\gamma) & = & \texttt{ide}!\ \mathrm{H}\,\gamma? \\
\texttt{take}^{\alpha}(\mathrm{T}\,\gamma) & = & \texttt{ide}!\ \mathrm{L}\,\gamma? \\
\texttt{take}^{\alpha}(\mathrm{L}\,\gamma) & = & \texttt{ide}!\ \mathrm{L}\,\gamma?
\end{array}$$

Hence, `take` is classified as lift strict in the first argument and not lift strict in the second. Again, this is not an approximation of the real behaviour, since `take 0` $\perp \Rightarrow$ `[]`. The situation changes if the empty list is not acceptable as a result, which implies that the argument list is definitely accessed (second and third equation). Note that tail strictness is not preserved, since the result is generally a true prefix of the argument: `take 2 (1:2:`$\perp$`)` $\Rightarrow$ `1:2:[]`.

`drop` The function `drop` is the counterpart of `take` as it returns the argument list without an initial segment of a given length: `drop 7` x returns all but the first seven elements.

$$
\begin{array}{lcl}
\mathtt{drop}^\alpha(\mathrm{B}) & = & \mathtt{bot}!\,\mathrm{B}! \\
\mathtt{drop}^\alpha(\mathrm{PH}\,\gamma) & = & \mathtt{ide}!\,\mathrm{PL}\,\gamma! \\
\mathtt{drop}^\alpha(\mathrm{PL}\,\gamma) & = & \mathtt{ide}!\,\mathrm{PL}\,\gamma! \\
\mathtt{drop}^\alpha(\mathrm{HT}\,\gamma) & = & \mathtt{ide}!\,\mathrm{T}\,\gamma! \\
\mathtt{drop}^\alpha(\mathrm{H}\,\gamma) & = & \mathtt{ide}!\,\mathrm{L}\,\gamma! \\
\mathtt{drop}^\alpha(\mathrm{T}\,\gamma) & = & \mathtt{ide}!\,\mathrm{T}\,\gamma! \\
\mathtt{drop}^\alpha(\mathrm{L}\,\gamma) & = & \mathtt{ide}!\,\mathrm{L}\,\gamma!
\end{array}
$$

It follows that `drop` is lift strict in both arguments. Contrary to `take` tail strictness is preserved while head strictness is lost. This is due to the fact that the result is usually a true suffix of the argument: `drop 2 (`$\perp$`:`$\perp$`:3:4:[])` $\Rightarrow$ `3:4:[]`.

Identity on lists The functions `take` and `drop` satisfy $x =$ `append (take` $n\ x$`) (drop` $n\ x$`)` for all natural numbers n and all lists x. Thus a very inefficient way to define the identity function on lists is the following.

```
idlist :: [α] → [α]
idlist x = append (take 61194 x) (drop 61194 x)
```

It is immediate that the least context transformer for the identity function is $\mathtt{idlist}^\alpha(\kappa) = \kappa!$. However, due to the use of independent and uniform contexts a much weaker context transformer is actually computed.

$$
\begin{array}{lcl}
\mathtt{idlist}^\alpha(\mathrm{B}) & = & \mathrm{B}! \\
\mathtt{idlist}^\alpha(\mathrm{PH}\,\gamma) & = & \mathrm{L}\,\gamma? \\
\mathtt{idlist}^\alpha(\mathrm{PL}\,\gamma) & = & \mathrm{L}\,\gamma? \\
\mathtt{idlist}^\alpha(\mathrm{HT}\,\gamma) & = & \mathrm{T}\,\gamma! \\
\mathtt{idlist}^\alpha(\mathrm{H}\,\gamma) & = & \mathrm{L}\,\gamma? \\
\mathtt{idlist}^\alpha(\mathrm{T}\,\gamma) & = & \mathrm{T}\,\gamma! \\
\mathtt{idlist}^\alpha(\mathrm{L}\,\gamma) & = & \mathrm{L}\,\gamma?
\end{array}
$$

The worst approximation occurs in the case of a head strict context where the ingoing information is completely lost. It probably stresses the obvious if we state that the fidelity of a context transformer naturally depends on the complexity of the function definition.

Top-down mergesort The top-down variant of mergesort[3] is based on the divide and conquer principle: Using `take` and `drop` the argument list is divided into two segments of equal length. Both of these are sorted via recursive calls to mergesort. Finally, the two resulting lists are merged (whence the name of the algorithm) producing a sorted permutation of the argument.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathtt{sort}^\alpha(\mathrm{B}) & = & \mathrm{B}! & \mathtt{sort}^\alpha(\mathrm{PH}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{sort}^\alpha(\mathrm{PL}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{sort}^\alpha(\mathrm{PL}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{sort}^\alpha(\mathrm{HT}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{bot}! & \mathtt{sort}^\alpha(\mathrm{HT}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
 & & & \mathtt{sort}^\alpha(\mathrm{H}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{sort}^\alpha(\mathrm{T}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{sort}^\alpha(\mathrm{T}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{sort}^\alpha(\mathrm{L}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{sort}^\alpha(\mathrm{L}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}!
\end{array}
$$

[3] Note that this variant is contained, for example, in the standard environment of MIRANDA.

According to the context transformer `sort` is lift strict and tail strict. On the negative side head strictness is not preserved which is clearly a rather crude approximation of the real behaviour. The imprecision is in some sense due to the use of `take` and `drop` (cf to the last example). The information that the concatenation of `take` n x and `drop` n x yields x again gets lost because the context transformer for `take` and `drop` do not take the value of n into account (`bot` and `ide` are the only contexts on `num`). The following 'experiment' illustrates the point further: Assume for the sake of the argument that the last line of the definition of `sort` is changed to `merge (sort (take (n2-1) x)) (sort (drop n2 x))`, that is, `take` takes one element less than in the original definition. Since the analysis works independent of any particular numerical value the same context transformer as for `sort` is derived. But now the loss of head strictness does not involve approximations, since the modified function omits some elements of its argument list.

Bottom-up mergesort The bottom-up variant of `mergesort` works as follows: In a first step the input list is mapped to a list of singleton lists. This list is then repeatedly traversed thereby merging adjacent elements until only a single list remains which represents the sorted permutation of the input list.

$$
\begin{array}{lcllcl}
\mathtt{msort}^{\alpha}(\mathrm{B}) & = & \mathrm{B}! & \mathtt{msort}^{\alpha}(\mathrm{PH}\,\mathtt{ide}) & = & \mathrm{HT}\,\mathtt{ide}! \\
\mathtt{msort}^{\alpha}(\mathrm{PL}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{msort}^{\alpha}(\mathrm{PL}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{msort}^{\alpha}(\mathrm{HT}\,\mathtt{bot}) & = & \mathrm{HT}\,\mathtt{bot}! & \mathtt{msort}^{\alpha}(\mathrm{HT}\,\mathtt{ide}) & = & \mathrm{HT}\,\mathtt{ide}! \\
 & & & \mathtt{msort}^{\alpha}(\mathrm{H}\,\mathtt{ide}) & = & \mathrm{HT}\,\mathtt{ide}! \\
\mathtt{msort}^{\alpha}(\mathrm{T}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{msort}^{\alpha}(\mathrm{T}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{msort}^{\alpha}(\mathrm{L}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{msort}^{\alpha}(\mathrm{L}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}!
\end{array}
$$

While `msort` denotes the same function as `sort` and is even based on the same sorting algorithm, its context transformer is strictly smaller than the one of `sort`. If the demand on the result contains a head strict component, then the complete input list is definitely required (second, fourth and fifth equation). It should be stressed that the other cases do not involve approximations, since `msort` does not inspect the list element(s) if the input list is a singleton: `msort [⊥] ⇒ [⊥]`.

`min` The minimum function, which determines the last element of a list of numerical values, may be defined in terms of a sorting routine.

```
min :: [num] → num
min x = head (msort x)
```

Note that this definition is as efficient as the obvious recursive one due to the use of lazy evaluation. Composing the context transformers of `head` and `msort` we obtain

$$
\begin{array}{lcl}
\mathtt{min}^{\alpha}(\mathtt{bot}) & = & \mathrm{B}! \\
\mathtt{min}^{\alpha}(\mathtt{ide}) & = & \mathrm{HT}\,\mathtt{ide}!.
\end{array}
$$

Hence, the complete argument list may be evaluated prior to the call of `min`.

Quicksort Another popular sorting algorithm which is based on the divide and conquer principle is quicksort. It is interesting to note that the context transformer of both variants given in Section B.3 is identical. It lies midway between $\mathtt{msort}^\alpha$ and $\mathtt{sort}^\alpha$.

$$\begin{array}{lcl@{\qquad}lcl}
\mathtt{qsort}^\alpha(\mathrm{B}) & = & \mathrm{B}! & \mathtt{qsort}^\alpha(\mathrm{PH}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{qsort}^\alpha(\mathrm{PL}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{qsort}^\alpha(\mathrm{PL}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{qsort}^\alpha(\mathrm{HT}\,\mathtt{bot}) & = & \mathrm{HT}\,\mathtt{bot}! & \mathtt{qsort}^\alpha(\mathrm{HT}\,\mathtt{ide}) & = & \mathrm{HT}\,\mathtt{ide}! \\
 & & & \mathtt{qsort}^\alpha(\mathrm{H}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{qsort}^\alpha(\mathrm{T}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{qsort}^\alpha(\mathrm{T}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{qsort}^\alpha(\mathrm{L}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{qsort}^\alpha(\mathrm{L}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}!
\end{array}$$

Approximations must be stated in the case of the head strict contexts PH `ide` and H `ide` which result only in a head strict rather than a head and tail strict context. Again, the inaccuracy may be attributed to the divide phase since the behaviour of the auxiliary function `partition`, which partitions a list with respect to a given pivot element, is modeled only imperfectly by the respective context transformer. Unfortunately, we have to refrain from listing $\mathtt{partition}^\alpha$ as it comprises 442 entries.

Primes The nullary function `primes` yields an infinite list containing the prime numbers in ascending order. Though `primes` has an empty list of arguments its context transformer is revealing.

$$\begin{array}{lcl@{\qquad}lcl}
\mathtt{primes}^\alpha(\mathrm{B}) & = & \mathtt{bot} & \mathtt{primes}^\alpha(\mathrm{PH}\,\mathtt{ide}) & = & \mathtt{ide} \\
\mathtt{primes}^\alpha(\mathrm{PL}\,\mathtt{bot}) & = & \mathtt{ide} & \mathtt{primes}^\alpha(\mathrm{PL}\,\mathtt{ide}) & = & \mathtt{ide} \\
\mathtt{primes}^\alpha(\mathrm{HT}\,\mathtt{bot}) & = & \mathtt{bot} & \mathtt{primes}^\alpha(\mathrm{HT}\,\mathtt{ide}) & = & \mathtt{bot} \\
 & & & \mathtt{primes}^\alpha(\mathrm{H}\,\mathtt{ide}) & = & \mathtt{ide} \\
\mathtt{primes}^\alpha(\mathrm{T}\,\mathtt{bot}) & = & \mathtt{bot} & \mathtt{primes}^\alpha(\mathrm{T}\,\mathtt{ide}) & = & \mathtt{bot} \\
\mathtt{primes}^\alpha(\mathrm{L}\,\mathtt{bot}) & = & \mathtt{ide} & \mathtt{primes}^\alpha(\mathrm{L}\,\mathtt{ide}) & = & \mathtt{ide}
\end{array}$$

The transformer does not convey any information about strictness it rather encodes termination properties. The interpretation is straightforward: If $\mathtt{primes}^\alpha(\kappa)$ yields `bot`, then the demand κ cannot be met, that is, if the context of `primes` contains a tail strict component, then the computation is doomed to fail.

7.6.4 Functions on trees

`size` The function `size` calculates the number of nodes in a given tree. Since the size corresponds closely to the `length` function, we obtain a similar context transformer.

$$\begin{array}{lcl}
\mathtt{size}^\alpha(\mathtt{bot}) & = & \mathrm{B}! \\
\mathtt{size}^\alpha(\mathtt{ide}) & = & \mathrm{LR}\,\mathtt{bot}!
\end{array}$$

The context LR `bot` shows that the 'structure' of the tree is definitely required while the elements of the tree are ignored which is completely analogous to the behaviour of `length`.

Inorder traversal There are several tree traversal algorithms which differ significantly in speed and in the number of '`:`' operations. However, all three variants listed in Appendix B possess the same context transformer.

$$
\begin{array}{lcl}
\texttt{inorder}^{\alpha}(\mathrm{B}) & = & \mathrm{B}! \\
\texttt{inorder}^{\alpha}(\mathrm{PH}\,\gamma) & = & \mathrm{L}\,\gamma! \\
\texttt{inorder}^{\alpha}(\mathrm{PL}\,\gamma) & = & \mathrm{L}\,\gamma! \\
\texttt{inorder}^{\alpha}(\mathrm{HT}\,\gamma) & = & \mathrm{LNR}\,\gamma! \\
\texttt{inorder}^{\alpha}(\mathrm{H}\,\gamma) & = & \mathrm{L}\,\gamma! \\
\texttt{inorder}^{\alpha}(\mathrm{T}\,\gamma) & = & \mathrm{LR}\,\gamma! \\
\texttt{inorder}^{\alpha}(\mathrm{L}\,\gamma) & = & \mathrm{L}\,\gamma!
\end{array}
$$

Since the first element of the output list is the leftmost element in the tree, `inorder` is strict and left strict. If `inorder` is embedded in a tail strict context, then the complete tree is traversed (sixth equation). If the context is additionally head strict, the elements of the tree are required as well. Note that isolated head strictness has no further effect as the head of the output list does in general not coincide with the root of the tree (contrary to the preorder traversal).

Preorder traversal Inorder and preorder traversal of a tree differ only in the order in which the elements labelled to the nodes are put in the output list. This difference is reflected nicely in the respective context transformers. [Again, all three variants listed in Appendix B give rise to the same context transformer.]

$$
\begin{array}{lcl}
\texttt{preorder}^{\alpha}(\mathrm{B}) & = & \mathrm{B}! \\
\texttt{preorder}^{\alpha}(\mathrm{PH}\,\gamma) & = & \mathrm{N}\,\gamma! \\
\texttt{preorder}^{\alpha}(\mathrm{PL}\,\gamma) & = & \mathrm{T}\,\gamma! \\
\texttt{preorder}^{\alpha}(\mathrm{HT}\,\gamma) & = & \mathrm{LNR}\,\gamma! \\
\texttt{preorder}^{\alpha}(\mathrm{H}\,\gamma) & = & \mathrm{N}\,\gamma! \\
\texttt{preorder}^{\alpha}(\mathrm{T}\,\gamma) & = & \mathrm{LR}\,\gamma! \\
\texttt{preorder}^{\alpha}(\mathrm{L}\,\gamma) & = & \mathrm{T}\,\gamma!
\end{array}
$$

Situated in a head strict context `preorder` needs the root element of its argument. If the context is tail strict `preorder` becomes left and right strict. Furthermore, both effects are independent of each other. Hence, the preorder traversal is in a sense more natural than the inorder traversal. Note that the argument may not be partial even if only a partial output list is required (second and third equation): Let $t = \texttt{Node Empty 1 (Node Empty 2 Empty)}$, then $\mathrm{PL}(\texttt{inorder}\,t) \Rightarrow \texttt{1:2:}\bot$, but $\texttt{inorder}(\mathrm{PT}\,t) \Rightarrow \texttt{1:}\bot$.

We conclude with a closer look at one particular implementation of the preorder traversal. The last variant listed in Appendix B employs an auxiliary function called `flatten`, which uses a stack of trees for bookkeeping purposes. Since the stack is realized by a list $\texttt{flatten}^{\alpha}$ serves as a nice example for nested polymorphic contexts.

$$
\begin{array}{lcl}
\texttt{flatten}^{\alpha}(\mathrm{B}) & = & \mathrm{B}! \\
\texttt{flatten}^{\alpha}(\mathrm{PH}\,\gamma) & = & \mathrm{PH}\,(\mathrm{N}\,\gamma)! \\
\texttt{flatten}^{\alpha}(\mathrm{PL}\,\gamma) & = & \mathrm{PH}\,(\mathrm{T}\,\gamma)! \\
\texttt{flatten}^{\alpha}(\mathrm{HT}\,\gamma) & = & \mathrm{HT}\,(\mathrm{LNR}\,\gamma)! \\
\texttt{flatten}^{\alpha}(\mathrm{H}\,\gamma) & = & \mathrm{H}\,(\mathrm{N}\,\gamma)! \\
\texttt{flatten}^{\alpha}(\mathrm{T}\,\gamma) & = & \mathrm{HT}\,(\mathrm{LR}\,\gamma)! \\
\texttt{flatten}^{\alpha}(\mathrm{L}\,\gamma) & = & \mathrm{H}\,(\mathrm{T}\,\gamma)!
\end{array}
$$

Hence, `flatten` is both lift strict and head strict. Furthermore the transformer tells us that the stack is completely worked off if the context is tail strict. The embedded contexts on trees correspond exactly to the demands inferred for `preorder`.

Top-down heapsort The most elaborate sorting algorithm is probably heapsort which uses balanced binary trees for fast access to the minimum of a bag of elements. The context transformer of the top-down variant is as good as it can be, that is, $\mathtt{hsort}^{\alpha}$ is identical to $\mathtt{msort}^{\alpha}$.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathtt{hsort}^{\alpha}(\mathrm{B}) & = & \mathrm{B}! & \mathtt{hsort}^{\alpha}(\mathrm{PH}\,\mathtt{ide}) & = & \mathrm{HT}\,\mathtt{ide}! \\
\mathtt{hsort}^{\alpha}(\mathrm{PL}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{hsort}^{\alpha}(\mathrm{PL}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{hsort}^{\alpha}(\mathrm{HT}\,\mathtt{bot}) & = & \mathrm{HT}\,\mathtt{bot}! & \mathtt{hsort}^{\alpha}(\mathrm{HT}\,\mathtt{ide}) & = & \mathrm{HT}\,\mathtt{ide}! \\
 & & & \mathtt{hsort}^{\alpha}(\mathrm{H}\,\mathtt{ide}) & = & \mathrm{HT}\,\mathtt{ide}! \\
\mathtt{hsort}^{\alpha}(\mathrm{T}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{hsort}^{\alpha}(\mathrm{T}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}! \\
\mathtt{hsort}^{\alpha}(\mathrm{L}\,\mathtt{bot}) & = & \mathrm{T}\,\mathtt{ide}! & \mathtt{hsort}^{\alpha}(\mathrm{L}\,\mathtt{ide}) & = & \mathrm{T}\,\mathtt{ide}!
\end{array}
$$

Bottom-up heapsort The bottom-up variant of heapsort differs from the top-down version only in the construction of the heap. Upon inspection we see that the bottom-up construction amounts to a smart realization of the divide and conquer scheme (for lists). This partly explains why the context transformer of `bhsort` is quite weak. It coincides, in fact, with $\mathtt{sort}^{\alpha}$.

Assessment: It seems that both bottom-up mergesort and top-down heapsort are more natural for the sorting of *lists* than algorithms based on the divide and conquer principle. Top-down mergesort, quicksort, and bottom-up heapsort are somehow tailored towards the sorting of *arrays*.[4] This completes our analysis of sorting algorithms.

7.6.5 A propositional theorem prover

The propositional theorem prover is the first more substantial program listed in Appendix B. The program is somehow typical for functional programming: The original problem is divided into comparatively small, manageable subproblems each of which is solved by a couple of functions.

The separation of concerns is supported by the ease with which new data types may be introduced. Using the type `dtree` α, for example, the construction of a deduction tree (the logic) is neatly separated from the traversal of the tree (the control). Lazy evaluation guarantees that this separation works in practice.

The analysis of the theorem prover, however, is hardly feasible due to the size of the intermediate types. Let us support this statement by some counts. The number of principal

[4]Let us support this statement by some further remarks. Bottom-up mergesort as well as top-down heapsort proceed in a strict left to right fashion respecting the inherent sequentiality of lists whereas the divide and conquer technique implies a non-sequential access to the elements of a list as indicated by the use of the auxiliary functions `take`, `drop` and `append`. It is interesting to note that these functions are not required in an implementation based on arrays. The following table further develops this point.

	array based		list based	
	mergesort	quicksort	mergesort	quicksort
divide	—	$O(n^2)$	$O(n \log n)$	$O(n^2)$
conquer	$O(n \log n)$	—	$O(n \log n)$	$O(n^2)$
	number of movements		number of ‘`:`’ operations	

Either the divide or the conquer phase comes for free in an array based implementation. Due to the use of `take` and `append` this does not hold for a list based implementation shedding some light on the shortcomings of top-down mergesort and quicksort. Note, however, that quicksort may be improved using accumulating parameters (see `qsortaux`). Furthermore, the recursive scheme employed in the bottom-up construction of a heap (see `buildn`) suggests a way of improving the divide phase of top-down mergesort.

contexts on the data type of propositional formulae is determined below.

$$\begin{array}{lrr}
\mu\beta.\text{Atom } \texttt{num}?\ | & 4 & 1 \\
\quad \text{Not } \beta?\ | & 2 & 1 \\
\quad \text{And } \beta?\ \beta?\ | & 4 & 3 \\
\quad \text{Or } \beta?\ \beta?\ | & 4 & 3 \\
\quad \text{Then } \beta?\ \beta?\ | & 4 & 3 \\
\quad \text{Iff } \beta?\ \beta? & 4 & 3 \\
\hline
 & 2.048 - & 81 + 1 = 1968
\end{array}
\qquad
\begin{array}{lr}
\mu\beta.\text{Leaf } \alpha?\ | & 2 \\
\quad \text{Inner } \texttt{[}\beta\texttt{]}? & 14 \\
\hline
 & 28
\end{array}$$

The total number of contexts for each construct is listed in the first column. The number of contexts which evaluate to $\perp\!\!\!\perp$ (if β is set to $\perp\!\!\!\perp$) is given in the second column. The calculation for the data type of deduction trees is a bit since there is no principal context on Leaf α? which evaluates to bottom.

The number of principal contexts on the type `([form],[num],[form],[num])` which is employed to represent a sequent is even more impressing. Since there are $6 \cdot 2 - 1 = 11$ principal contexts on `[num]` and $6 \cdot 1968 - 1 = 11807$ on `[form]`, we have a total of $(2 \cdot 11807 - 1)^2(2 \cdot 11 - 1)^2 + 1 = 245.890.032.130$ principal contexts on `sequent`.

Now, note that the auxiliary function `expand` has the result type `[sequent]`. Hence its context transformer comprises $6 \cdot 245.890.032.130 - 1 = 1.475.340.192.779$ equations which makes explicit why we refrain from analysing the theorem prover on basis of the technique presented in this chapter.

It would be interesting, however, to carry out the analysis using minimal function graphs (cf to [105]) since the context transformers of the main functions `taut` and `dnf` are comparatively small: $\texttt{taut}^\alpha$ consists of 4 and $\texttt{dnf}^\alpha$ of 1968 equations.

7.6.6 A pretty-printer for first-order expressions

Let us first determine the number of principal contexts on `text` and `expr` α.

$$\begin{array}{lrr}
\mu\beta.\text{Str } \texttt{[char]}?\ | & 22 & 1 \\
\quad \text{Block } \texttt{[}\beta\texttt{]}?\ | & 14 & 2 \\
\quad \text{Indent } \beta?\ | & 2 & 1 \\
\quad \text{NL} & 2 & 1 \\
\hline
 & 1.232 - & 2 + 1 = 1231
\end{array}$$

Due to the nested type structure there are quite a few principal contexts on `expr` α (not to speak of `expr [char]` which could be the result type of a parser for first-order expressions).

$$\begin{array}{lr}
\mu\beta.\text{Var } \alpha?\ | & 2 \\
\quad \text{Freeze } \beta?\ | & 2 \\
\quad \text{Unfreeze } \beta?\ | & 2 \\
\quad \text{Sys } \alpha?\ \beta?\ | & 4 \\
\quad \text{Con } \alpha?\ \beta?\ | & 4 \\
\quad \text{Fun } \alpha?\ \beta?\ | & 4 \\
\quad \text{Sequ } \texttt{[}\beta\texttt{]}?\ | & 14 \\
\quad \text{Case } \beta?\ \texttt{[(}\alpha\texttt{,[}\alpha\texttt{],}\beta\texttt{)]}?\ | & 1267 \\
\quad \text{Let } \texttt{[}\alpha\texttt{]}?\ \beta?\ \beta?\ | & 56 \\
\quad \text{Rec } \texttt{[}\alpha\texttt{]}?\ \beta?\ \ | & 27 \\
\hline
 & 13.731.766.272
\end{array}$$

The huge number of contexts illustrates impressively the problems an implementation of backward analysis has to cope with.

Though the context transformers for the main functions `flatten` and `showexpr` both comprise only 11 equations we will not specify them for lack of space (the argument types are too large). The curious reader is put off until Section 8.6.5 where the results are presented in a compact form.

7.7 Run-time complexity

Let us conclude the chapter with estimations of the expected run-time complexity. We assume that the function to be analysed is explicitly typed, monomorphic and does not contain recursive local definitions. The requirement that the function is explicitly typed simplifies the analysis of the operations $\sqcup^\alpha$ and $\&^\alpha$. Note that the explicitly typed program may be exponentially bigger than the underlying typed program. Consider, for example,

$$a_1 \texttt{ = (}a_2\texttt{,}a_2\texttt{); } a_2 \texttt{ = (}a_3\texttt{,}a_3\texttt{); ... ; } a_{n-1} \texttt{ = (}a_n\texttt{,}a_n\texttt{); } a_n \texttt{ = (0908,1995),}$$

which is of size $\Theta(n)$ as an implicitly typed program but of size $\Theta(2^n)$ as an explicitly typed one. However, programs arising in practice are unlikely to exhibit such a worst-case behaviour (the types of top-level definitions are often specified anyway for documentary purposes).

The analysis of a function $f :: \sigma_1 \to \sigma_2$ consists essentially of four nested 'loops': The context transformer of f is obtained by a fixpoint iteration the length of which is bounded by the height of the function lattice $\|\sigma_2\| \to \|\sigma_1\|$. Recall that the length of the space of monotone functions $D \to_m E$ is in turn bounded by length$(D \to_m E) \leqslant$ card$(D)\cdot$length(E). In each iteration step a context transformer is calculated, that is, for each context in $\|\sigma_2\|$ the expression tree of the function's body is traversed. Finally, at each node an operation is performed ($\sqcup^\alpha$, $\&^\alpha$, fold, unfold etc) whose running time is either proportional to the size of the types of the free variables or to the size of the type of the expression. Since the program is explicitly typed their running time is bounded by the size of the program. To summarize: Let s be the size of f's definition, let $m =$ size(σ_1) and $n =$ size(σ_2), then we have by Fact 6.26

1. length of fixpoint iteration: $\mathrm{O}(m \cdot 2^n)$,
2. number of principal contexts: $\mathrm{O}(2^n)$,
3. traversal of the expression tree: $\mathrm{O}(s)$,
4. cost of primitive operations: $\mathrm{O}(s)$,

leading to a worst-case complexity of $\mathrm{O}(m \cdot 4^n \cdot s^2)$. A few remarks are necessary: The estimation of the length of the fixpoint iteration is probably very imprecise since we made only use of the fact that a context transformer is monotone. However, by Lemma 5.4 we know that the *least abstraction* of a function is additive, that is, $\varphi^\sharp(\pi_1 \sqcup \pi_2) = \varphi^\sharp(\pi_1) \sqcup \varphi^\sharp(\pi_2)$. If f^α were additive as well the height would be bounded by $m \cdot n$, see [123], for example. Unfortunately, due to the use of independent projections respectively contexts additivity does not hold for the analysis. However, we do not expect a context transformer to deviate so grossly from its ideal, so the actual length of a fixpoint iteration is probably closer to $m \cdot n$ as to $m \cdot 2^n$. This conjecture is also supported by the practical experiments listed in Section 8.7. Accordingly, the cost of the primitive operations is much closer to m as to s.

The running time of the analysis is, in fact, dominated by the size of the context transformer the upper bound of which is quite precise (if we ignore the normalization of contexts the bound is even exact). Hence, to obtain strictness analysis results in an acceptable time further improvements are necessary. The next chapter presents a new technique which aims at this direction.

Turning to the analysis of polymorphic functions we note that the length of the fixpoint iteration and the size of the context transformer are further influenced by the number of type variables occurring in the *result type* of f. Recall that the principal contexts are essentially generated by 'replacing' each type variable in σ_2 by a fresh context variable. Let $\{\alpha_1, \ldots, \alpha_i\}$ be the *set* of type variables in σ_2 and let $\Gamma_2 = \{\gamma_1 : \|\alpha_1\|, \ldots, \gamma_i : \|\alpha_i\|\}$, then the number of principal contexts is bounded by size$(\Gamma_2 \vdash \sigma_2)$. It is noteworthy that size$(\Gamma_2 \vdash \sigma_2)$ is actually smaller than the size of the context lattice for the simplest monomorphic instance of σ_2. There are, for example, 7 principal contexts on $[\alpha]$ but 11 on `[num]`. On the negative side the codomain of the context transformer is generally much larger since the number of occurrences of a type variable must be taken into account. More precisely, let $\wr \alpha'_1, \ldots, \alpha'_j \wr$ be the *multiset* of the type variables in σ_2 and let $\Gamma_1 = \{\gamma'_1 : \|\alpha'_1\|, \ldots, \gamma'_j : \|\alpha'_j\|\}$, then the height of the codomain is bounded by size$(\Gamma_1 \vdash \sigma_2)$. However, since it is quite exceptional that the same type variable occurs more than three or four times in the result type of a function, the additional level of complexity has little implications in practice (cf to the remark after Lemma 6.10).

7.8 Bibliographic notes

Hughes [81] was the first to deal with the backward analysis of polymorphic functions. Building on the view of first-order polymorphic functions as natural transformations he shows in *loc. cit.* that a polymorphic function is characterized by a single instance (actually some further conditions are required). The practical method he proposes is, in fact, a special case of ours: It is obtained by replacing the context variables on the right hand side by their least upper bounds (separately for each type variable). The analysis of `dup`, for example, only yields $\mathtt{dup}^{\alpha}(\gamma_1!, \gamma_2!) = (\gamma_1 + \gamma_2)!$ etc.

Building on this work Kubiak *et. al.* [105] developed a prototype implementation for a first-order subset of HASKELL. This chapter is in parts based on their paper. In *loc. cit.* a non-standard semantics and an approximation semantics are sketched. However, no proofs of correctness or safety are given. Hence, this chapters continues the preceding one in putting [105] on a formal basis.

Chapter 8

Generic Strictness Analysis

Our projection for strictness analysis is a rosy future.
— P. Wadler and R.J.M. Hughes, *Projections for Strictness Analysis (1987)*

We have alluded several times to the problem that the size of a context transformer grows exponentially with respect to the size of the underlying type. If a strictness analyser is to become an integral part of a compiler exponential run-time behaviour is hardly acceptable. This chapter presents a technique, termed generic analysis, with a dramatically improved average-case behaviour. Compared to the straightforward realization of backward analysis described in the last chapter this technique is orders of magnitude faster—Section 8.7 presents some timings.

An alternative approach described in the literature involves minimal function graphs [105] which are due to N.D. Jones and A. Mycroft [96]. Minimal function graphs allow to compute only a part of a context transformer. This approach is appropriate for local function definitions since the number of contexts in which a local function is situated is fixed (and usually small compared to the number of principal contexts). Thus the whole context transformer is rarely required. The technique is, however, not of much help in connection with separate compilation (if separate compilation is taken seriously). It seems to be unavoidable to compute the whole context transformer of a global function and to record it in the interface of a module. Albeit using the classical analysis both undertakings are equally unrealistic due to the size of a context transformer. R. Kubiak *et. al.* [105] write

> "The most serious problem comes from modules. If the strictness analyser is to return results, strictness properties have to be passed from one module to another. Currently, it far from clear how to do this."

Let us tackle the problem of compact representation first since it paves the way for the generic analysis. Furthermore, let us restrict ourselves to *monomorphic* functions. Since a context amounts to a type expression decorated with strictness annotations, the contexts appearing on the left and right hand sides of a context transformer differ only in the respective strictness annotations.

$$\begin{aligned}
\texttt{member}^{\alpha}(\text{False}\ \texttt{bot} \mid \text{True}\ \texttt{bot}) &= (\mu\beta.\texttt{[]}\ \texttt{bot} \mid \texttt{bot!:}\beta!)!\ \texttt{bot!}\\
\texttt{member}^{\alpha}(\text{False}\ \texttt{bot} \mid \text{True}\ \texttt{ide}) &= (\mu\beta.\texttt{[]}\ \texttt{bot} \mid \texttt{ide!:}\beta?)!\ \texttt{ide!}\\
\texttt{member}^{\alpha}(\text{False}\ \texttt{ide} \mid \text{True}\ \texttt{bot}) &= (\mu\beta.\texttt{[]}\ \texttt{ide} \mid \texttt{ide!:}\beta!)!\ \texttt{ide?}\\
\texttt{member}^{\alpha}(\text{False}\ \texttt{ide} \mid \text{True}\ \texttt{ide}) &= (\mu\beta.\texttt{[]}\ \texttt{ide} \mid \texttt{ide!:}\beta?)!\ \texttt{ide?}
\end{aligned}$$

Recall that we have two kinds of annotations: the two contexts `bot` and `ide` on system-defined types ordered by $\texttt{bot} \lesssim \texttt{ide}$ and the two lifts ordered by $! \lesssim ?$. In each case we have a lattice isomorphic to the lattice of truth values. Now, if the contexts specifying the demand on the function's result contain m strictness annotations and the contexts specifying the demand on the argument to the function contain n annotations, the context transformer corresponds to a Boolean function of type $\mathbb{B}^m \to \mathbb{B}^n$. Furthermore, the truth function is monotone since the context transformer is. Here is the representation of $\texttt{member}^\alpha$ which is of type $\mathbb{B}^2 \to \mathbb{B}^7$.

$$\texttt{member}^\alpha(\text{False } a \mid \text{True } b) \quad = \quad (\mu\beta.\texttt{[]}\ a \mid (a \vee b)!:\beta b)!\ (a \vee b)a$$

If we remove the type components the transformer reads as $\Phi(a,b) = (a, a \vee b, !, b, !, a \vee b, a)$. The letters a and b stand for propositional variables, the monotone truth functions are represented by positive Boolean terms (terms build up from propositional variables using the connectives $\wedge$ and $\vee$). Note that we continue to use the symbols `bot`, `ide`, '!' and '?' which now act as truth constants!

When interpreting such a context transformer it is important to keep in mind that Boolean terms are used for both contexts and lifts, false representing `bot` and '!', true representing `ide` and '?'. The transformer $\texttt{member}^\alpha$ may be interpreted as follows: First note that `False` is not acceptable as a result if a is set to false. Accordingly, if b is set to false then `True` is not acceptable as a result. Hence, the subterm `[]` a means that the empty list is not acceptable as an argument iff `False` is not acceptable as a result (`member [] a` $\Rightarrow$ `False`). The Boolean term $a \vee b$ ensures that the bottom context is propagated. The lift of the second argument is strict iff `False` is not acceptable as a result. Finally, `member` is lift and head strict in the first argument irrespective of the given demand.

The central idea of the generic analysis is to employ contexts decorated with Boolean terms already during the fixpoint iteration. Instead of computing the images of all principal contexts only the image of *the principal generic context* is determined. The latter term denotes the context appearing on the left hand side of the context transformers above. The principal generic context on `bool` is False $a \mid$ True b, for $\texttt{[}\alpha\texttt{]}$ we have $\mu\beta.\texttt{[]}\ a \mid \gamma b:\beta c$, that is, subcontexts on system-defined types and lifts are set to different propositional variables.

It is relatively easy to generalize the operations employed in the classical analysis to this representation. Consider, for example, the guard operation which is listed below (let us assume that the 'guarded' context is of lift type).

$$\begin{aligned} !\rhd^\alpha \kappa\ell &= \kappa\ell \\ ?\rhd^\alpha \kappa\ell &= \kappa\ell \sqcup^\alpha \texttt{bot}? \quad = \quad \kappa? \end{aligned}$$

Since we interpret '!' by false and '?' by true the guard rule boils down a a logical disjunction of Boolean terms.

$$p \rhd^\alpha \kappa q \quad = \quad \kappa(p \vee q)$$

The conjunction and disjunction of contexts on primitive types may be modified accordingly.

$$\varsigma p \sqcup^\alpha \varsigma q \quad = \quad \varsigma(p \vee q) \qquad\qquad \varsigma p \mathbin{\&^\alpha} \varsigma q \quad = \quad \varsigma(p \wedge q)$$

The operation which cost the authors a few grey cells is the conjunction of contexts on lifts[1] whose definition is repeated below.

$$\begin{aligned} \kappa! \mathbin{\&^\alpha} \kappa'! &= (\kappa \mathbin{\&^\alpha} \kappa')! \\ \kappa! \mathbin{\&^\alpha} \kappa'? &= (\kappa \sqcup^\alpha \kappa \mathbin{\&^\alpha} \kappa')! \\ \kappa? \mathbin{\&^\alpha} \kappa'! &= (\kappa \mathbin{\&^\alpha} \kappa' \sqcup^\alpha \kappa')! \\ \kappa? \mathbin{\&^\alpha} \kappa'? &= (\kappa \sqcup^\alpha \kappa')? \end{aligned}$$

[1]The author worked already just under a year on the topic until he discovered the appropriate definition. It occurred to him as he tried to illustrate the approximations a previous set of rules involved.

Problems are caused by the non-uniform behaviour of $\&^\alpha$. If both lifts are given by non-trivial Boolean terms it seems that we do not have enough information as to decide which right-hand side should be considered next. [The first solution to this problem was to assume the worst possible case and to set $\kappa p \ \&^\alpha\ \kappa' q \ = \ (\kappa \sqcup^\alpha \kappa')(p \wedge q)$. While this is obviously safe it is also very imprecise as one may imagine.] The solution of this problem becomes apparent if we organize the right hand-sides more systematically using the law of inclusion ($\pi_1 \sqcup \pi_2 = \pi_1 \sqcup \pi_1 \ \&\ \pi_2 \sqcup \pi_2$).

$$
\begin{array}{rcl}
\kappa! \ \&^\alpha\ \kappa'! & = & (\qquad \kappa \ \&^\alpha\ \kappa' \qquad)! \\
\kappa! \ \&^\alpha\ \kappa'? & = & (\kappa \sqcup^\alpha \kappa \ \&^\alpha\ \kappa' \qquad)! \\
\kappa? \ \&^\alpha\ \kappa'! & = & (\qquad \kappa \ \&^\alpha\ \kappa' \sqcup^\alpha \kappa')! \\
\kappa? \ \&^\alpha\ \kappa'? & = & (\kappa \sqcup^\alpha \kappa \ \&^\alpha\ \kappa' \sqcup^\alpha \kappa')?
\end{array}
$$

Now, the first disjunctive term $\kappa \sqcup^\alpha \cdot$ disappears if the lift of the second argument is strict. The presence of the third disjunctive term $\cdot \sqcup^\alpha \kappa'$ depends accordingly on the lift of the first argument. But how do we proceed if we do not know the concrete values of the lifts? The solution is surprisingly simple: Let $\kappa \wedge p$ denote the context in which each Boolean term in κ is conjugated with p, then the conjunction may be defined by

$$\kappa p \ \&^\alpha\ \kappa' q \quad = \quad (q \wedge \kappa \sqcup^\alpha \kappa \ \&^\alpha\ \kappa' \sqcup^\alpha \kappa' \wedge p)(p \wedge q).$$

The original equations are obtained by instantiating p and q to strict and lazy lift respectively. For example, if p is set to false and q to true, we have $q \wedge \kappa = \kappa$ and $\kappa \wedge p = \texttt{bot}$. Consequently $\kappa! \ \&^\alpha\ \kappa'? = (\kappa \sqcup^\alpha \kappa \ \&^\alpha\ \kappa')!$ as desired.

Turning to polymorphic functions things become more complicated as contexts now presumably also differ in their polymorphic components. The context transformer of `dup` is a good example. Recall that $\texttt{dup}^\alpha$ essentially paraphrases the behaviour of $\&$ on lifts.

$$
\begin{array}{rcl}
\texttt{dup}^\alpha(\gamma_1!, \gamma_2!) & = & (\gamma_1 \ \texttt{\&}\ \gamma_2)! \\
\texttt{dup}^\alpha(\gamma_1!, \gamma_2?) & = & (\gamma_1 \ \texttt{+}\ \gamma_1 \ \texttt{\&}\ \gamma_2)! \\
\texttt{dup}^\alpha(\gamma_1?, \gamma_2!) & = & (\gamma_1 \ \texttt{\&}\ \gamma_2 \ \texttt{+}\ \gamma_2)! \\
\texttt{dup}^\alpha(\gamma_1?, \gamma_2?) & = & (\gamma_1 \ \texttt{+}\ \gamma_2)?
\end{array}
$$

The solution to this problem profits, of course, from the discussion above. The central idea is to introduce a syntactic variant of $\kappa \wedge p$, thus delaying this operation until instances of the context variables contained in the polymorphic context κ are known. The context transformer of `dup` may then be condensed to

$$\texttt{dup}^\alpha(\gamma_1 a, \gamma_2 b) \quad = \quad (b \wedge \gamma_1 \ \texttt{+}\ \gamma_1 \ \texttt{\&}\ \gamma_2 \ \texttt{+}\ \gamma_2 \wedge a)(a \wedge b).$$

This little extension suffices in fact to develop a generic variant of the theory of polymorphic contexts.

Plan of the chapter Section 8.1 introduces generic contexts generalizing the concepts given in Chapter 6. The changes to the auxiliary functions and to the semantic equations are presented in Sections 8.2, 8.3, 8.4, and 8.5 (it is quite surprising how few modifications are required). The examples of Section 7.6 are re-considered using the generic analysis in Section 8.6. Section 8.7 concludes with a discussion of the expected run-time behaviour and presents some benchmarks.

Since the generic analysis generalizes the classical analysis we shall specify only the required modifications. We will stick in particular to the notation and to the nomenclature introduced in the two previous chapters.

8.1 Generic contexts

This section is essentially structured according to Chapter 6. In order to keep the chapter self-contained Section 8.1.1 reviews the basic facts about positive Boolean terms. [Since we will never need terms with negation we will sometimes be sloppy and omit the qualifier 'positive'.] Section 8.1.2 develops the theory of generic polymorphic contexts. The results presented are then generalized to contexts on arbitrary types. Section 8.1.3 is concerned with syntax and semantics. Section 8.1.5 introduces a normalform for generic contexts which is used to characterize order and equality syntactically. Finally, Section 8.1.5 shows how to transfer disjunction and conjunction to generic contexts.

8.1.1 Positive Boolean terms

The syntactic categories of positive Boolean terms are shown in Table 8.1.

meta variable		category	comment
a, b, c	$\in$	**bvar**	Boolean variables
p, q	$\in$	**bexp**	positive Boolean terms

Table 8.1: Syntactic categories of positive Boolean terms

8.1 Definition *Let* $V \subseteq$ **bvar** *be a set of Boolean variables. Then* p *is a* positive Boolean term with variables in V *iff* $V \vdash p :$ **bexp** *is derivable using the deduction rules below.*

$$\frac{}{V \vdash 0 : \mathbf{bexp}} \qquad \frac{}{V \vdash 1 : \mathbf{bexp}} \qquad \frac{}{V \cup \{a\} \vdash a : \mathbf{bexp}}$$

$$\frac{V \vdash p : \mathbf{bexp} \quad V \vdash q : \mathbf{bexp}}{V \vdash p \vee q : \mathbf{bexp}} \qquad \frac{V \vdash p : \mathbf{bexp} \quad V \vdash q : \mathbf{bexp}}{V \vdash p \wedge q : \mathbf{bexp}}$$

The set of all positive Boolean terms wrt V *is denoted by* $\mathbf{bexp}(V)$.

As it is customary we will often abbreviate the logical conjunction $p \wedge q$ by pq. Furthermore, it is understood that the conjunction has a higher precedence than the disjunction: $a \wedge b \vee c$ is an abbreviation for $(a \wedge b) \vee c$.

The semantics of a Boolean term is given by Table 8.2. We say that p logically implies q (notation: $p \models q$) iff $(\forall \eta)\ \mathcal{B}[\![p]\!]\eta \sqsubseteq \mathcal{B}[\![q]\!]\eta$ where the truth values are ordered by $\mathbf{f} \sqsubseteq \mathbf{t}$ (ex falso quod libet). The terms p and q are called logically equivalent (notation: $p \simeq q$) iff $(\forall \eta)\ \mathcal{B}[\![p]\!]\eta = \mathcal{B}[\![q]\!]\eta$.

It is well-known that every monotone map $\phi : \mathbb{B}^n \to \mathbb{B}$ may be represented by some positive Boolean term in n Boolean variables. Conversely, every positive Boolean term defines a monotone map. Since the set of monotone maps from $\mathbb{B}^n$ to $\mathbb{B}$ ordered pointwise forms a finite lattice, so does $\langle \mathbf{bexp}(V_n)/\simeq; \models \rangle$ with $V_n = \{a_1, \ldots, a_n\}$. Elements of $\mathbf{bexp}(V_n)$ may be canonically represented using minimal polynoms which are introduced subsequently (see Section 8.7 for more advanced techniques).

Every positive Boolean term may be converted to a disjunction of conjunctions by repeatedly applying the distributive laws. Furthermore, the ACI properties of $\wedge$ and $\vee$ suggest

Let η be an assignment mapping Boolean variables to truth values, then $\mathcal{B}[\![p]\!]\eta$ denotes the truth value of p with respect to the assignment η.

$$\begin{array}{rcl}
\mathcal{B}[\![\mathbf{bexp}]\!] & : & (\mathbf{bvar} \to \mathbb{B}) \to \mathbf{bexp} \to \mathbb{B} \\
\mathcal{B}[\![a]\!]\eta & = & \eta(a) \\
\mathcal{B}[\![0]\!]\eta & = & \mathbf{f} \\
\mathcal{B}[\![1]\!]\eta & = & \mathbf{t} \\
\mathcal{B}[\![p \wedge q]\!]\eta & = & \mathcal{B}[\![p]\!]\eta \wedge \mathcal{B}[\![q]\!]\eta \\
\mathcal{B}[\![p \vee q]\!]\eta & = & \mathcal{B}[\![p]\!]\eta \vee \mathcal{B}[\![q]\!]\eta
\end{array}$$

Table 8.2: The semantics of Boolean terms

to represent expression in disjunctive normalform by sets of sets of Boolean variables. The set

$$\{\{a_{11}, \ldots, a_{1n_1}\}, \ldots, \{a_{m1}, \ldots, a_{mn_m}\}\} \subseteq \wp_{\mathrm{f}}(\wp_{\mathrm{f}}(V))$$

represents the expression

$$a_{11} \wedge \ldots \wedge a_{1n_1} \vee \ldots \vee a_{m1} \wedge \ldots \wedge a_{mn_m}.$$

The truth constant 0 is represented by the empty set, the truth constant 1 by $\{\varnothing\}$. As is stands the disjunctive normalform is not unambiguous: a and $a \vee ab$ are equivalent and both are in disjunctive normalform. An unambiguous form is obtained by repeatedly applying the law of absorption $\phi_1 \vee (\phi_1 \wedge \phi_2) = \phi_1$. If we remove all redundant conjuncts we obtain minimal polynoms. [Note that the difference between the algebraic theory of polymorphic contexts and of positive Boolean terms results essentially from the fact that the inclusion law $\pi_1 \sqcup \pi_2 = \pi_1 \sqcup \pi_1 \mathbin{\&} \pi_2 \sqcup \pi_2$ is weaker than the absorption law.]

8.2 Definition and Lemma *Let $V \subseteq$ **bvar**, the function **m** : $\wp_{\mathrm{f}}(\wp_{\mathrm{f}}(V)) \to \wp_{\mathrm{f}}(\wp_{\mathrm{f}}(V))$ with*

$$\mathbf{m}(\mathcal{A}) \quad = \quad \{\, A \mid A \in \mathcal{A} \wedge \neg(\exists B \in \mathcal{A})\; B \subset A \,\}$$

maps its argument to the disjunctive normalform. An element of $\mathbf{m}(\wp_{\mathrm{f}}(\wp_{\mathrm{f}}(V)))$ is called minimal polynom. *The set of all minimal polynoms with respect to V is denoted $\mathbf{dnf}(V)$. Let $V_n := \{a_1, \ldots, a_n\}$ be a set of Boolean variables, then $\langle \mathbf{dnf}(V_n);\ \subseteq \rangle$ forms a finite lattice.*

The lattices $\mathbf{dnf}(V_n)$ for $1 \leqslant n \leqslant 3$ are pictured in Figure 8.1. The lattice $\mathbf{dnf}(V_4)$ is already to big to be drawn properly as it comprises 168 points. Note that $\mathbf{dnf}(V_n)$ is considerably smaller than the lattice of polymorphic contexts $\mathbf{cnf}(C_n)$.

The conjunction and disjunctions of minimal polynoms is defined below.

8.3 Definition *Let $\mathcal{A}, \mathcal{B} \in \mathbf{bexp}(V)$ be minimal polynoms, the logical conjunction and disjunction of minimal polynoms is defined by*

$$\begin{array}{rcl}
\mathcal{A} \veebar \mathcal{B} & = & \mathbf{m}(\mathcal{A} \cup \mathcal{B}), \\
\mathcal{A} \barwedge \mathcal{B} & = & \mathbf{m}\{\, A \cup B \mid A \in \mathcal{A} \wedge B \in \mathcal{B} \,\}.
\end{array}$$

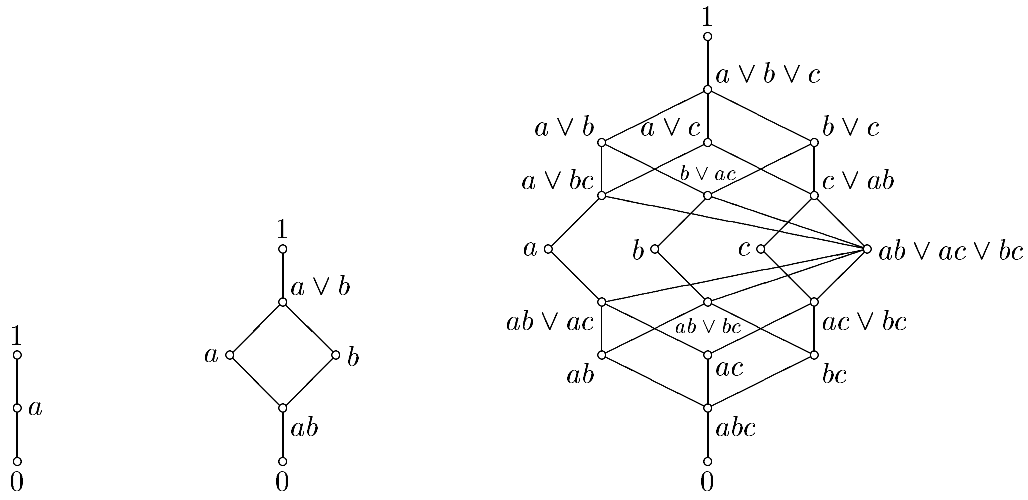


Figure 8.1: The lattices $\mathbf{dnf}(V_n)$ for $1 \leqslant n \leqslant 3$

The function norm : $\mathbf{bexp}(V) \to \mathbf{dnf}(V)$ *with*

$$\begin{aligned}
\text{norm}[\![a]\!] &= \{\{a\}\} \\
\text{norm}[\![0]\!] &= \varnothing \\
\text{norm}[\![1]\!] &= \{\varnothing\} \\
\text{norm}[\![p \vee q]\!] &= \text{norm}[\![p]\!] \veebar \text{norm}[\![q]\!] \\
\text{norm}[\![p \wedge q]\!] &= \text{norm}[\![p]\!] \barwedge \text{norm}[\![q]\!]
\end{aligned}$$

maps its argument to the disjunctive normalform.

8.1.2 Generic polymorphic contexts

We have motivated in the introduction to this chapter that the only extension necessary is the conjunction of a polymorphic context with a Boolean term $\kappa \wedge p$, called *guarded context*. As we shall see in the sequel this extension fits nicely in the theory developed in Section 6.1. The algebraic program becomes by now probably a matter of routine. We start by defining syntax and semantics and then introduce normalform and order.

8.4 Definition *Let $C \subseteq$ **cvar** be a set of context variables and let $V \subseteq$ **bvar** be a set of Boolean variables. Then κ is a* generic polymorphic context with respect to C and V *iff $C, V \vdash \kappa :$ **cnt** is derivable using the deduction rules below.*

$$\frac{}{C, V \vdash \texttt{bot} : \mathbf{cnt}} \qquad \frac{}{C \cup \{\gamma\}, V \vdash \gamma : \mathbf{cnt}}$$

$$\frac{C, V \vdash \kappa : \mathbf{cnt} \quad C, V \vdash \kappa' : \mathbf{cnt}}{C, V \vdash \kappa \mathbin{\&} \kappa' : \mathbf{cnt}} \qquad \frac{C, V \vdash \kappa : \mathbf{cnt} \quad C, V \vdash \kappa' : \mathbf{cnt}}{C, V \vdash \kappa + \kappa' : \mathbf{cnt}}$$

$$\frac{C, V \vdash \kappa : \mathbf{cnt} \quad V \vdash p : \mathbf{bexp}}{C, V \vdash \kappa \wedge p : \mathbf{cnt}}$$

The set of all polymorphic contexts wrt C and V is denoted by $\mathbf{cnt}(C, V)$*.*

The guarded context $\kappa \wedge p$ is sometimes also written as $p \wedge \kappa$. The precedence of $\wedge$ lies midway between `&` and `+`. Hence, γ_1 + γ_1 & $\gamma_2 \wedge p$ is an abbreviation for γ_1 + $((\gamma_1$ & $\gamma_2) \wedge p)$.

We shall try, of course, to profit as much as possible from the results presented in Section 6.1. As a first step towards this end the semantics of generic polymorphic contexts is specified by transforming a generic context to a polymorphic context according to a given assignment of truth values to Boolean variables. If η is such an assignment then let $\eta(\kappa)$ be syntax for the evaluation of κ wrt η. The guarded context $\kappa \wedge p$ evaluates to `bot` if p denotes false and to κ otherwise. The formal definition of $\eta(\kappa)$ is given in Table 8.3.

Let η be an assignment mapping Boolean variables to truth values. Then $\eta(\kappa)$ denotes the polymorphic context obtained by evaluating the guarded contexts in κ with respect to η.

$$\begin{array}{rcll} \eta(\texttt{bot}) & = & \texttt{bot} & \\ \eta(\gamma) & = & \gamma & \\ \eta(\kappa \mathbin{\&} \kappa') & = & \eta(\kappa) \mathbin{\&} \eta(\kappa') & \\ \eta(\kappa + \kappa') & = & \eta(\kappa) + \eta(\kappa') & \\ \eta(\kappa \wedge p) & = & \texttt{bot} & \textbf{if } \mathcal{B}[\![p]\!]\eta = \mathbf{f} \\ & = & \eta(\kappa) & \textbf{otherwise} \end{array}$$

Table 8.3: Transformation of a generic polymorphic context to a polymorphic context

The following definition generalizes the semantic relations $\approx$ and $\lessapprox$ (cf Definition 6.2) to generic polymorphic contexts.

8.5 Definition *Let $C \subseteq$ **cvar**, let $V \subseteq$ **bvar** and let $\kappa, \kappa' \in$ **cnt**(C, V) be generic polymorphic contexts. Then*

$$\begin{array}{lcl} \kappa \approx \kappa' & :\Longleftrightarrow & (\forall \eta)\ \eta(\kappa) \approx \eta(\kappa'), \\ \kappa \lessapprox \kappa' & :\Longleftrightarrow & (\forall \eta)\ \eta(\kappa) \lessapprox \eta(\kappa'). \end{array}$$

Recall from Section 6.1 that each element of the lattice **cnt**$(C_n)/\approx$ can be represented by a disjunction of conjunctions. Now, using the guarded context $\kappa \wedge p$ we may represent, for example, the elements of **cnt**$(C_2)/\approx$ uniformly by the term $\gamma_1 \wedge a$ + γ_1 & $\gamma_2 \wedge (ab \vee c)$ + $\gamma_2 \wedge b$. By varying the assignment η we obtain each polymorphic context in **cnt**(C_2). Even more general each *generic* polymorphic context in **cnt**(C_2, V) is convertible into the form $\gamma_1 \wedge p$ + γ_1 & $\gamma_2 \wedge (pq \vee r)$ + $\gamma_2 \wedge q$ which is therefore a suitable candidate for a normalform. The presence of each conjunct is given by a Boolean term: p determines the presence of γ_1, q that of γ_2 and r that of γ_1 & γ_2. Furthermore the polymorphic context which is obtained by applying an assignment η is automatically in normalform. This is managed by the disjunctive Boolean term pq which ensures that γ_1 & γ_2 is present if both γ_1 and γ_2 are.

The normalforms of generic and standard contexts are closely related. The normalform of the latter may be interpreted as a mapping $\wp^\circ_{\mathrm{f}}(C) \to \mathbb{B}$ (satisfying the closure condition of Definition 6.6) which specifies whether a conjunction represented by a non-empty set is contained in the disjunction. The generic normalform amounts to a mapping $\wp^\circ_{\mathrm{f}}(C) \to$ **dnf**(V) where the presence of a conjunction is given by a Boolean term. The closure operation **n** is mimicked accordingly using logical operations.

The following definition introduces the formal calculus on which the equational reasoning is based.

8.6 Definition [Extension of Definition 6.4] *The relations $\lesssim$ and $\sim$ on polymorphic contexts are defined by the axioms and rules of Definition 6.4 extended by the following set.*

$$\begin{aligned}
\mathtt{bot} \wedge p &\sim \mathtt{bot} \\
\kappa \wedge 0 &\sim \mathtt{bot} \\
\kappa \wedge 1 &\sim \kappa \\
(\kappa_1 + \kappa_2) \wedge p &\sim \kappa_1 \wedge p + \kappa_2 \wedge p \\
\kappa \wedge p + \kappa \wedge q &\sim \kappa \wedge (p \vee q) \\
(\kappa_1 \wedge p) \mathbin{\&} (\kappa_2 \wedge p) &\sim \kappa \mathbin{\&} \kappa_2 \wedge (p \vee q) \\
(\kappa \wedge p) \wedge q &\sim \kappa \wedge (p \wedge q)
\end{aligned}
\qquad
\frac{p \simeq q}{\kappa \wedge p \sim \kappa \wedge q}$$

The following recipe may be used to convert an arbitrary polymorphic context into its normalform.

1. Push the guards $\cdot \wedge p$ to the leaves of the expression tree.
2. Using the distributive laws push in the conjunctions.
3. Bring the guards to the top of the conjunctions and unite guarded contexts with identical conjunctions.
4. Insert missing conjunctions and missing guards.

The results of this procedure is a full disjunction of guarded conjunctions. It is not difficult to see that this term corresponds to a function of type $\wp^\circ_{\mathrm{f}}(C) \to \mathbf{dnf}(V)$. The normalform is then obtained by repeatedly applying the law of inclusion and $\kappa \wedge p + \kappa \wedge q \sim \kappa \wedge (p \vee q)$. This operation can be modeled by a closure operator akin to the one introduced in Definition 6.6.

8.7 Definition and Fact *Let $C \subseteq$ **cvar** and let $V \subseteq$ **bvar**, then the function $\mathbf{n} : (\wp^\circ_{\mathrm{f}}(C) \to \mathbf{dnf}(V)) \to (\wp^\circ_{\mathrm{f}}(C) \to \mathbf{dnf}(V))$ with*

$$\mathbf{n}(\Phi) \;=\; \boldsymbol{\lambda} A. \underline{\bigvee}\{\, \overline{\bigwedge}\{\, \Phi(B) \mid B \in \mathcal{B} \,\} \mid \bigcup \mathcal{B} = A \,\}$$

is a closure operator. An element of $\mathbf{n}(\wp^\circ_{\mathrm{f}}(C) \to \mathbf{dnf}(V))$ is termed normal generic polymorphic context. *The set of all normal generic polymorphic contexts wrt C and V is denoted by $\mathbf{cnf}(C, V)$.*

Note that the definition of $\mathbf{n}$ may be obtained by rephrasing Definition 6.6 in terms of characteristic functions instead of sets. The difference is, in fact, that the logical operations are now interpreted syntactically. Therefore we have the following

8.8 Fact *Let Φ be a generic polymorphic context, then*

$$(\forall \eta)\ \eta(\mathbf{n}(\Phi)) = \mathbf{n}(\eta(\Phi)).$$

As usual we build upon the normalform of contexts. For that reason let us define variants of $\mathbin{\&}^\alpha$, $\sqcup^\alpha$ and $\overline{\wedge}$ which work directly on normalforms.

8.9 Definition *Let $\Phi, \Psi \in \mathbf{cnf}(C, V)$ be normal generic polymorphic contexts. The operations $\mathbin{\&}^\alpha$, $\sqcup^\alpha$, and $\overline{\wedge}$ are defined by*

$$\begin{aligned}
\Phi \mathbin{\&}^\alpha \Psi &= \boldsymbol{\lambda} A. \underline{\bigvee}\{\, \Phi(B) \mathbin{\overline{\wedge}} \Psi(C) \mid B \cup C = A \,\}, \\
\Phi \sqcup^\alpha \Psi &= \boldsymbol{\lambda} A. \Phi(A) \mathbin{\underline{\vee}} (\Phi \mathbin{\&}^\alpha \Psi)(A) \mathbin{\underline{\vee}} \Psi(A), \\
\Phi \mathbin{\overline{\wedge}} p &= \boldsymbol{\lambda} A. \Phi(A) \mathbin{\overline{\wedge}} p.
\end{aligned}$$

The function norm *with*

$$\begin{array}{rcl}
\text{norm}[\![\texttt{bot}]\!] &=& \boldsymbol{\lambda}A.0\\
\text{norm}[\![\gamma]\!] &=& \boldsymbol{\lambda}A.(A=\{\gamma\}\longrightarrow 1 \mid 0)\\
\text{norm}[\![\kappa \,\&\, \kappa']\!] &=& \text{norm}[\![\kappa]\!] \mathbin{\&^\alpha} \text{norm}[\![\kappa']\!]\\
\text{norm}[\![\kappa + \kappa']\!] &=& \text{norm}[\![\kappa]\!] \sqcup^\alpha \text{norm}[\![\kappa']\!]\\
\text{norm}[\![\kappa \wedge p]\!] &=& \text{norm}[\![\kappa]\!] \mathbin{\bar{\wedge}} \text{norm}[\![p]\!]
\end{array}$$

maps its argument to the context normalform.

The normalforms for generic and standard contexts are related by

8.10 Fact *Let κ be a generic polymorphic context, then*

$$(\forall\eta)\ (\eta(\text{norm}[\![\kappa]\!]) \quad = \quad \text{norm}(\eta(\kappa))).$$

The Adequacy Theorem generalizes to generic contexts.

8.11 Adequacy Theorem for generic polymorphic contexts *Let $\kappa, \kappa' \in$ $\mathbf{cnt}(C_n, V_m)$ be generic polymorphic contexts, then*

$$\begin{array}{lllll}
\kappa \approx \kappa' & \Longleftrightarrow & \kappa \sim \kappa' & \Longleftrightarrow & \text{norm}[\![\kappa]\!] = \text{norm}[\![\kappa']\!],\\
\kappa \lessapprox \kappa' & \Longleftrightarrow & \kappa \lesssim \kappa' & \Longleftrightarrow & \text{norm}[\![\kappa]\!] \sqsubseteq \text{norm}[\![\kappa']\!].
\end{array}$$

Hence, $\langle\mathbf{cnt}(C_n, V_m)/{\approx};\ \lessapprox\rangle$, $\langle\mathbf{cnt}(C_n, V_m)/{\sim};\ \lesssim\rangle$ *and* $\langle\mathbf{cnf}(C_n, V_m);\ \sqsubseteq\rangle$ *are isomorphic finite lattices.*

Proof We show the two propositions simultaneously. $(2) \Longrightarrow (1)$: It is not hard to see that the rules are correct. $(3) \Longrightarrow (2)$: Follows by $\text{norm}(\kappa) \sim \kappa$. $(1) \Longrightarrow (3)$:

$$\begin{array}{rcll}
\kappa \approx \kappa' & \Longrightarrow & (\forall\eta)\ \eta(\kappa) \approx \eta(\kappa') & \text{def.} \approx\\
& \Longrightarrow & (\forall\eta)\ (\text{norm}(\eta(\kappa)) = \text{norm}(\eta(\kappa'))) & \text{Adequacy Theorem 6.9}\\
& \Longrightarrow & (\forall\eta)\ (\eta(\text{norm}[\![\kappa]\!]) = \eta(\text{norm}[\![\kappa']\!])) & \text{Fact 8.10}\\
& \Longrightarrow & \text{norm}[\![\kappa]\!] = \text{norm}[\![\kappa']\!] &
\end{array}$$

Analogous for '$\lesssim$'. ∎

8.1.3 Syntax and semantics

To avoid unnecessary repetitions of the material presented in Section 6.2 we will specify only the modifications to the respective definitions.

8.12 Definition [Modification of Definition 6.11] *Let $\sigma \in$ **texp** be a type, let Γ be a set of assumptions of the form $\gamma : \|\alpha\|$ with $\gamma \in$ **cvar** and $\alpha \in$ **tvar**, and let $V \subseteq$ **bvar** be a set of Boolean variables. We say that κ is a* generic context on σ wrt Γ and V *iff $\Gamma, V \vdash \kappa : \|\sigma\|$ is derivable using the deduction rules below.*

3. $$\frac{V \vdash p : \mathbf{bexp}}{\Gamma, V \vdash \varsigma p : \|\varsigma\|} \qquad (\varsigma)$$

4. $$\frac{\Gamma, V \vdash \kappa : \|\sigma\| \quad V \vdash p : \mathbf{bexp}}{\Gamma, V \vdash \kappa p : \|\sigma ?\|} \qquad (\sigma ?)$$

The set of all generic contexts on σ wrt Γ and V is denoted by $|\Gamma, V \vdash \sigma|$.

It is understood that the cases which are not explicitly listed above are to be found in the original definition. Some simple examples for generic contexts are the following: False a | True b is an element of $|\varnothing, V \vdash \texttt{bool}|$ with $V = \{a, b\}$ and $\mu\beta.\texttt{[]}\ a \mid \gamma b{:}\beta c$ is an element of $|\Gamma, V \vdash \texttt{[}\alpha\texttt{]}|$ with $\Gamma = \{\gamma : \|\alpha\|\}$ and $V = \{a, b, c\}$.

As in the case of polymorphic contexts we specify the semantics by giving a transformation to contexts in the sense of Definition 6.11. The context ςp is mapped to $\varsigma\texttt{bot}$ or $\varsigma\texttt{ide}$ according to the truth value of p. Accordingly, the context κp is mapped to the strict or to the lazy lift. Table 8.4 summarizes the transformation rules. Definition 8.5 which introduces the relations $\approx$ and $\lessapprox$ generalizes without modifications to generic contexts.

Let η be an assignment mapping Boolean variables to truth values. Then $\eta(\kappa)$ denotes the context obtained by evaluating the Boolean terms in κ according to η.

$$\begin{aligned}
\eta(\varsigma p) &= \varsigma\texttt{bot} && \textbf{if } \mathcal{B}[\![p]\!]\eta = \textbf{f} \\
&= \varsigma\texttt{ide} && \textbf{otherwise} \\
\eta(\kappa p) &= \eta(\kappa)! && \textbf{if } \mathcal{B}[\![p]\!]\eta = \textbf{f} \\
&= \eta(\kappa)? && \textbf{otherwise}
\end{aligned}$$

Table 8.4: Transformation of a generic context to a context

8.1.4 Normalform and order

This section generalizes the definitions and theorems of Section 6.3 to generic contexts.

Definition 8.13 introduces the canonical representation of the bottom context.

8.13 Definition [Modification of Definition 6.15] *Let σ be a type, then* $\mathrm{bot}(\sigma)$ *denotes the bottom context on σ.*

$$\begin{aligned}
\mathrm{bot}(\varsigma) &= \varsigma 0 \\
\mathrm{bot}(\sigma?) &= \mathrm{bot}(\sigma)0
\end{aligned}$$

The context $\mathrm{gen}(\sigma, p)$ *denotes either the disjunctive or the conjunctive unit depending on the value of p.*

$$\begin{aligned}
\mathrm{gen}(\sigma?, p) &= \mathrm{bot}(\sigma)p \\
\mathrm{gen}(\varepsilon, p) &= \varepsilon p \\
\mathrm{gen}(\sigma_1 \dots \sigma_n, p) &= \mathrm{gen}(\sigma_1, p) \dots \mathrm{gen}(\sigma_n, p)
\end{aligned}$$

The generic context $\mathrm{gen}(\sigma, p)$ is, for example, employed in the definition of the guard operation. Therefore we postpone a discussion until Section 8.3.

A generic context κ is in normalform iff $\eta(\kappa)$ is normal in the sense of Definition 6.17, for all assignments η. The normalform of $\texttt{bot}a\ \texttt{bot}b$ and $\texttt{bot}(a \vee b)\ \texttt{bot}(ab)$, for example, is $\texttt{bot}(ab)\ \texttt{bot}(ab)$. The normalform is computed as follows: First a propositional formula is calculated which exactly identifies the cases in which the given context is proper. This can be done by replacing the logical connectives in the function 'improper' by their syntactic counterparts (well, for technical reasons we must actually work with the complement of 'improper'). The desired formula for both $\texttt{bot}a\,\texttt{bot}b$ and $\texttt{bot}(a \vee b)\,\texttt{bot}(ab)$ is ab, since the contexts are proper if and only if ab evaluates to true. The formula is then conjugated with each Boolean term contained in the given context yielding $\texttt{bot}(ab)\,\texttt{bot}(ab)$ in our running example.

8.14 Definition *The function* proper *converts a generic context into a positive Boolean term which evaluates to false iff the given context denotes the bottom projection.*

$$\begin{array}{rcl}
\text{proper}[\![\alpha\kappa]\!]\bar{\varrho} & = & 0 \quad \textbf{if } \text{norm}[\![\kappa]\!] = \texttt{bot} \\
& = & 1 \quad \textbf{otherwise} \\
\text{proper}[\![\beta]\!]\bar{\varrho} & = & \bar{\varrho}(\beta) \\
\text{proper}[\![\varsigma p]\!]\bar{\varrho} & = & p \\
\text{proper}[\![\kappa p]\!]\bar{\varrho} & = & \text{proper}[\![\kappa]\!]\bar{\varrho} \veebar p \\
\text{proper}[\![c_1\kappa_1 \mid \cdots \mid c_n\kappa_n]\!]\bar{\varrho} & = & \text{proper}[\![\kappa_1]\!]\bar{\varrho} \veebar \cdots \veebar \text{proper}[\![\kappa_n]\!]\bar{\varrho} \\
\text{proper}[\![\kappa_1 \ldots \kappa_n]\!]\bar{\varrho} & = & \text{proper}[\![\kappa_1]\!]\bar{\varrho} \barwedge \cdots \barwedge \text{proper}[\![\kappa_n]\!]\bar{\varrho} \\
\text{proper}[\![\mu\beta.\kappa]\!]\bar{\varrho} & = & \text{proper}[\![\kappa]\!](\bar{\varrho}[\beta \mapsto 0]).
\end{array}$$

Finally, $\text{proper}[\![\kappa]\!]$ *serves as an abbreviation for* $\text{proper}[\![\kappa]\!](\boldsymbol{\lambda}\beta.1)$.

8.15 Fact *Let* $\kappa \in |\Gamma, V \vdash \sigma|$ *be a generic context. For all assignments* $\bar{\varrho}$ *and* ϱ *such that* $\mathcal{B}(\bar{\varrho}(\beta)) = \neg\varrho(\beta)$ *the following holds*

$$(\forall\eta)\ \mathcal{B}(\text{proper}[\![k]\!]\bar{\varrho})\eta = \neg\,\text{improper}(\eta(\kappa))\,\varrho.$$

8.16 Definition [Modification of Definition 6.17] *The function* norm *maps a generic context to its* normalform.

$$\begin{array}{rcl}
\text{norm}[\![\varsigma p]\!] & = & \varsigma\text{norm}[\![p]\!] \\
\text{norm}[\![\kappa p]\!] & = & \text{norm}[\![\kappa]\!]\text{norm}[\![p]\!] \\
\text{norm}[\![\kappa_1 \ldots \kappa_n]\!] & = & (\text{norm}[\![\kappa_1]\!] \ldots \text{norm}[\![\kappa_n]\!]) \barwedge \text{proper}[\![\kappa_1 \ldots \kappa_n]\!] \\
\text{norm}[\![\mu\beta.\kappa]\!] & = & (\mu\beta.\text{norm}[\![\kappa]\!]) \barwedge \text{proper}[\![\mu\beta.\kappa]\!]
\end{array}$$

The generic context κ *is called* in normalform *iff* $\text{norm}[\![\kappa]\!] = \kappa$. *The set of all normal generic contexts on* σ *wrt* Γ *and* V *is denoted by* $\|\Gamma, V \vdash \sigma\|$.

8.17 Fact *Let* κ *be a generic context, then*

$$(\forall\eta)\ (\eta(\text{norm}[\![\kappa]\!]) \quad = \quad \text{norm}(\eta(\kappa))).$$

Let $\kappa = \mu\beta.\texttt{[]}\ a \mid \gamma b\,{:}\,\beta c$, then $\text{proper}[\![\kappa]\!] = a \vee c$, that is, κ is proper iff either the empty list is acceptable or else the demand on the recursive component is lazy. Consequently, the normalform of κ is $\mu\beta.\texttt{[]}\ a \mid \gamma(ab \vee ac)\,{:}\,\beta c$.

Restricted to contexts in normalform the order $\lesssim$ boils down to the coordinatewise order. Recall that the propositional formulae contained in generic contexts are ordered by logical entailment.

8.18 Definition [Modification of Definition 6.18] *The relation* $\lesssim$ *on normal generic contexts is defined by the following rules.*

3. $$\frac{p \models q}{\varsigma p \lesssim \varsigma q} \qquad (\varsigma)$$

4. $\dfrac{\kappa \lesssim \kappa' \quad p \models q}{\kappa p \lesssim \kappa' q}$ $(\sigma?)$

8.19 Adequacy Theorem for generic contexts *Let Γ be a finite set of assumptions, let V be a finite set of Boolean variables and let $\kappa, \kappa' \in \|\Gamma, V \vdash \sigma\|$ be normal generic contexts, then*

$$\begin{aligned} \kappa \approx \kappa' &\iff \kappa = \kappa', \\ \kappa \lessapprox \kappa' &\iff \kappa \lesssim \kappa'. \end{aligned}$$

Hence, $\langle \|\Gamma, V \vdash \sigma\|/{\approx}; \lessapprox \rangle$, and $\langle \|\Gamma, V \vdash \sigma\|; \lesssim \rangle$ are isomorphic finite orders.

8.1.5 Disjunction and conjunction

We have already commented on most of the changes to the definition of conjunction and disjunction. It remains to generalize the conjunction of recursive contexts. Recall that the conjunction of $\mu\beta.\kappa$ and $\mu\beta.\kappa'$ proceeds in two steps. First the least upper bound of the recursive calls in $\kappa \mathbin{\&^\alpha} \kappa'$ is determined. The resulting context then serves as a template for the conjunction of $\mu\beta.\kappa$ and $\mu\beta.\kappa'$. This procedure is essentially the same in the generic case with the slight difference that we now work with Boolean terms which specify the presence or absence of conjuncts. Suppose $q \wedge \gamma_1 + \gamma_1 \,\&\, \gamma_2 \wedge r + \gamma_2 \wedge p$ is the least upper bound, then we set $(\mu\beta.\kappa) \mathbin{\&^\alpha} (\mu\beta.\kappa') = \mu\beta.q \barwedge \kappa_1 \sqcup^\alpha \kappa_1 \mathbin{\&^\alpha} \kappa_2 \sqcup^\alpha \kappa_2 \barwedge p$.

8.20 Definition [Modification of Definition 6.21] *Conjunction and disjunction of generic contexts are defined as follows.*

3. $\varsigma p \sqcup^\alpha \varsigma q \;=\; \varsigma(p \veebar q) \qquad \varsigma p \mathbin{\&^\alpha} \varsigma q \;=\; \varsigma(p \barwedge q)$ (ς)

4. $\kappa p \sqcup^\alpha \kappa' q \;=\; (\kappa \sqcup^\alpha \kappa')(p \veebar q)$ $(\sigma?)$
$\kappa p \mathbin{\&^\alpha} \kappa' q \;=\; (q \barwedge \kappa \sqcup^\alpha \kappa \mathbin{\&^\alpha} \kappa' \sqcup^\alpha \kappa' \barwedge p)(p \barwedge q)$

7. $(\mu\beta.\kappa) \sqcup^\alpha (\mu\beta.\kappa') \;=\; \mu\beta.\kappa \sqcup^\alpha \kappa'$ $(\mu\beta.\sigma)$
$(\mu\beta.\kappa) \mathbin{\&^\alpha} (\mu\beta.\kappa')$
$= \;\mathrm{norm}(\mu\beta.q \barwedge \kappa \sqcup^\alpha \kappa \mathbin{\&^\alpha} \kappa' \sqcup^\alpha \kappa' \barwedge p)$
where
$q \wedge \gamma_1 + \gamma_1 \,\&\, \gamma_2 \wedge r + \gamma_2 \wedge p = (\mathrm{norm}(\kappa[\beta/\alpha\gamma_1]) \mathbin{\&^\alpha} \mathrm{norm}(\kappa[\beta/\alpha\gamma_2])) \updownarrow \alpha$

Let us consider some examples.

$$\begin{aligned} & (\mathtt{num}\,a)b \mathbin{\&^\alpha} (\mathtt{num}\,\mathtt{bot})c \\ = \;& (c \barwedge \mathtt{num}\,a \sqcup^\alpha \mathtt{num}\,a \mathbin{\&^\alpha} \mathtt{num}\,\mathtt{bot} \sqcup^\alpha \mathtt{num}\,\mathtt{bot} \barwedge b)bc \\ = \;& (\mathtt{num}\,ac \sqcup^\alpha \mathtt{num}\,\mathtt{bot} \sqcup^\alpha \mathtt{num}\,\mathtt{bot})bc \\ = \;& (\mathtt{num}\,ac)bc \end{aligned}$$

The reader may convince herself that the result is accurate by instantiating a, b, and c to contexts and lifts.

$$\begin{aligned} & (\mu\beta.\,\mathtt{[]}\ a \mid \gamma b : \beta!) \mathbin{\&^\alpha} (\mu\beta.\,\mathtt{[]}\ c \mid \gamma! : \beta d) \\ = \;& \mu\beta.d \barwedge (\mathtt{[]}\ a \mid \gamma b : \beta!) \sqcup^\alpha (\mathtt{[]}\ a \mid \gamma b : \beta!) \mathbin{\&^\alpha} (\mathtt{[]}\ c \mid \gamma! : \beta d) \\ = \;& \mu\beta.(\mathtt{[]}\ ad \mid \gamma bd : \beta!) \sqcup^\alpha (\mathtt{[]}\ ac \mid \gamma! : \beta!) \\ = \;& \mu\beta.\,\mathtt{[]}\ ac \vee ad \mid \gamma bd : \beta! \end{aligned}$$

The generic operations truly mimic the standard ones.

8.21 Fact *Let $\kappa, \kappa' \in \|\Gamma, V \vdash \sigma\|$ be normal generic contexts, then*

$$(\forall \eta)\ \eta(\kappa \sqcup^{\alpha} \kappa') = \eta(\kappa) \sqcup^{\alpha} \eta(\kappa'),$$
$$(\forall \eta)\ \eta(\kappa \,\&^{\alpha}\, \kappa') = \eta(\kappa) \,\&^{\alpha}\, \eta(\kappa').$$

This immediately implies that the disjunction implements the join operation.

8.22 Lemma *Let $\kappa, \kappa' \in \|\Gamma, V \vdash \sigma\|$ be normal generic contexts. Then $\kappa \sqcup^{\alpha} \kappa'$ is the join of κ and κ'. Hence, $\langle \|\Gamma, V \vdash \sigma\|; \lesssim \rangle$ forms a finite lattice.*

8.2 System- and user-defined functions

The context transformer of a system-defined function may be interpreted as a Boolean function of type $\mathbb{B} \to \mathbb{B}^{2m}$ where m is the arity of the respective primitive.

$$\begin{aligned} \texttt{+}^{\alpha}(\texttt{num}a) &= (\texttt{num}a)!\,(\texttt{num}a)! \\ \texttt{*}^{\alpha}(\texttt{num}a) &= (\texttt{num}a)!\,(\texttt{num}a)a \end{aligned}$$

As the major achievement of the generic analysis the context transformer of a user-defined function is tabulated only for the single principal generic context on the respective type. The principal generic context on `bool` is False a | True b, on $[\alpha]$ we have $\mu\beta.\texttt{[]}\ a \mid \gamma b : \beta c$. Formally, a principal context is the greatest element with respect to the subsumption ordering defined below. [As in Chapter 7 let us fix the set of assumptions Γ and the set of Boolean variables V. Then $\|\sigma\|$ serves as an abbreviation for $\|\Gamma, V \vdash \sigma\|$.]

8.23 Definition *Let $\kappa_1, \kappa_2 \in \sum \vartheta.\|\sigma\vartheta\|$ be normal generic contexts. Then κ_2* subsumes *κ_1 (notation: $\kappa_1 \preccurlyeq \kappa_2$) iff there is a substitution θ on context variables and a substitution ξ on Boolean variables sich that $\kappa_1 = \kappa_2\theta\xi$.*

8.24 Definition *Let $\kappa \in \sum \vartheta.\|\sigma\vartheta\|$ be a normal generic context. Then κ is termed* principal *iff $[\kappa]_{\equiv}$ is the greatest element with respect to $\preccurlyeq$.*

Since we have extended the set of polymorphic contexts Definition 7.4 which defines the evaluation of a context needs an extension as well.

8.25 Definition and Fact [Extension of Definition 7.4] *The function* eval *evaluates a generic polymorphic context with respect to given substitutions on context and Boolean variables.*

$$\text{eval}[\![\kappa \wedge p]\!]\theta\xi \quad = \quad \text{eval}[\![\kappa]\!]\theta\xi \mathbin{\overline{\wedge}} \text{norm}(p\xi).$$

The function eval *is generalized in the obvious way to normal generic contexts. For all generic contexts $\kappa \in \|\sigma\|$ and for all substitutions θ and ξ the following holds*

$$(\forall \eta)\ \eta(\text{eval}[\![\kappa]\!]\theta\xi) = \text{eval}(\eta(\kappa\xi))(\eta(\theta)).$$

The correctness of the generic analysis is proven relative to the standard analysis. More precisely, we show that a generic context transformer gives the same results for all assignments of truth values to Boolean variables. Or to put it another way, a generic context transformer is correct iff it is a compact representation of the corresponding standard one. The following definition formally introduces the correctness criterion.

8.26 Definition *Let* $\tau^\alpha : \prod \vartheta.\|\sigma_1\vartheta\| \to \|\sigma_2\vartheta\|$ *be a context transformer and let* $\bar{\tau}^\alpha$ *be a generic context transformer, then* $\bar{\tau}^\alpha$ *encompasses* τ^α *iff*

$$(\forall\eta)\ \eta(\bar{\tau}^\alpha(\hat{\kappa})) = \tau^\alpha(\eta(\hat{\kappa})),$$

where $\hat{\kappa}$ *is the principal generic context on* σ_2*.*

The application of a generic context transformer is straightforward.

8.27 Definition and Fact *Let* $\bar{\tau}^\alpha : \prod \vartheta.\|\sigma_1\vartheta\| \to \|\sigma_2\vartheta\|$ *be a generic context transformer, let* ϑ *be a type substitution and let* $\kappa \in \|\sigma_1\vartheta\|$ *be a generic context. Then* $\bar{\tau}^\alpha \cdot \kappa \in \|\sigma_2\vartheta\|$ *is defined by*

$$\bar{\tau}^\alpha \cdot \kappa \quad = \quad \mathrm{eval}(\bar{\tau}^\alpha(\hat{\kappa}))\theta\xi \ \textbf{where}\ \hat{\kappa}\theta\xi = \kappa.$$

Let τ^α *be a context transformer. If* $\bar{\tau}^\alpha$ *encompasses* τ^α*, then* $\boldsymbol{\lambda}\kappa.\bar{\tau}^\alpha \cdot \kappa$ *encompasses* $\boldsymbol{\lambda}\kappa.\tau^\alpha \cdot \kappa$*.*

8.3 Operations on lift types

Using the context $\mathrm{gen}(\sigma, p)$ defined in 8.13 the adaptation of the guard operator is particularly simple. Since $\mathrm{bot}(\sigma)$ is neutral with respect to $\sqcup^\alpha$ the definition of $\rhd^\alpha$ may be rewritten as

$$\begin{aligned} !\rhd^\alpha \kappa &= \kappa \sqcup^\alpha \mathrm{bot}(\sigma), \\ ?\rhd^\alpha \kappa &= \kappa \sqcup^\alpha \mathrm{abs}(\sigma). \end{aligned}$$

As $\mathrm{gen}(\sigma, p)$ is either $\mathrm{bot}(\sigma)$ or $\mathrm{abs}(\sigma)$ depending on the truth value of p we immediately have the following

8.28 Definition and Fact *Let* $\kappa \in \|\sigma\|$ *be a normal generic context and let* $p \in$ **dnf** *be a normal positive Boolean term. The context* $p \rhd^\alpha \kappa \in \|\sigma\|$ *is defined by*

$$p \rhd^\alpha \kappa \quad = \quad \kappa \sqcup^\alpha \mathrm{gen}(\sigma, p).$$

For all normal generic contexts $\kappa \in \|\sigma\|$ *and for all Boolean terms* p *the following holds*

$$(\forall\eta)\ \eta(p \rhd^\alpha \kappa) = \eta(p) \rhd^\alpha \eta(\kappa).$$

where $\eta(p) = !$ *if* $\mathcal{B}[\![p]\!]\eta = \mathbf{f}$ *and* $\eta(p) = ?$ *otherwise.*

8.4 Operations on sequence types

The definition of the operator $\langle\rangle^\alpha$ requires a slight amendment.

8.29 Definition and Fact *Let* $\kappa \in \|\varepsilon\|$ *be a normal generic context. The context* $\langle\rangle^\alpha(\kappa) \in \|\sigma\|$ *is defined by*

$$\langle\rangle^\alpha(\varepsilon p) \quad = \quad \mathrm{gen}(\sigma, p).$$

For all normal generic contexts $\kappa \in \|\varepsilon\|$ *and for all Boolean terms* p *the following holds*

$$(\forall\eta)\ \eta(\langle\rangle^\alpha(\kappa)) = \langle\rangle^\alpha(\eta(\kappa)).$$

8.5 Semantic domains and equations

To discriminate between the standard analysis and the generic one we consistently overline the entities of the latter one. The analysis of an expression, for example, is written $\bar{\mathcal{E}}^\alpha[\![e]\!]\ \bar{\phi}^\alpha$.

The modifications to the semantic equations listed in Table 7.4 are, in fact, minor. All we have to do is to replace the operations on contexts by their generic counterparts introduced in the last sections. For obvious reasons we refrain from listing Table 7.4 a second time. The proof of correctness is equally straightforward since the primitive operations faithfully mimic the standard ones.

8.30 Definition *Let ϕ^α be an approximating function environment and let $\bar{\phi}^\alpha$ be a generic function environment. Then $\bar{\phi}^\alpha$ encompasses ϕ^α iff $\bar{\phi}^\alpha(f)$ encompasses $\phi^\alpha(f)$, for all $f \in$* **fun**.

8.31 Correctness of the generic analysis

1. *If $\bar{\phi}^\alpha$ encompasses ϕ^α, then $\bar{\mathcal{E}}^\alpha[\![e_0]\!]\ \bar{\phi}^\alpha$ encompasses $\mathcal{E}^\alpha[\![e_0]\!]\ \phi^\alpha$ for all e_o :* **dbe**.
2. *$\bar{\mathcal{P}}^\alpha[\![f_1 = \lambda e_1; \ldots ; f_n = \lambda e_n]\!]$ encompasses $\mathcal{P}^\alpha[\![f_1 = \lambda e_1; \ldots ; f_n = \lambda e_n]\!]$.*

8.6 Examples

Generic analysis on the test bench. In Section 7.6 we have analysed a bunch of functions using the standard analysis. In the sequel we will re-analyse (most of) the examples employing the technique introduced in this chapter.

The analysis has been successfully implemented in Miranda.[2] The generic context transformers listed below represent essentially the output of the analyser. Minor modifications have been made as to enhance their readability (cf to the paragraph on idioms and shortcuts below).

The analyser is with one notable exception a true transcription of its mathematical definition. We decided *not* to employ the normalform given in Definition 8.16. The reasons for this decision are rather empirical: The constant restoration of the normalform is computationally very demanding while the gain in accuracy is comparatively small. The examples given below show that the approximations introduced are, in fact, negligible. Furthermore, contexts in normalform tend to be awkward to read due to the inflation of conjunctive terms.

For functions of type $\sigma \rightarrow \alpha$ where α is a type variable the two techniques make no difference. Hence we refrain from repeating the context transformers of `if`, `const` etc.

8.6.1 Simple functions

`sor` The generic context transformer of the 'sequential or' coincides with the standard one.

$$\texttt{sor}^\alpha(\text{False } a \mid \text{True } b) \quad = \quad (\text{False } \texttt{ide} \mid \text{True } b)!\ (\text{False } a \mid \text{True } b)b$$

Note though that the right hand side of the context transformer is not in normalform which reads as (False $(a \vee b)$ $\mid$ True b)! (False a $\mid$ True $b)b$. [It is quite amusing that the logical

[2] Again, the program owes much to an earlier implementation by Michael Kettler which is discussed at length in [102]. Note, however, that the analysis described in *loc. cit.* is based on a weaker set of rules. The conjunction of contexts on lifts, for example, is given by $\kappa p \mathbin{\&^\alpha} \kappa' q = (\kappa \sqcup^\alpha \kappa')(p \wedge q)$.

disjunction is used to characterize the computational behaviour of 'sequential or'.] The generic context transformer makes explicit that the demand on the result is propagated literally to the second argument and that the second argument is required if and only if `True` is not acceptable as a result.

8.6.2 Functions on lists

`append` Let us again work out the analysis of `append`. The principal generic context on $\texttt{[}\alpha\texttt{]}$ is given by $\Phi(\gamma; a, b, c) = \mu\beta.\texttt{[]}\ a \mid \gamma b{:}\beta c$. The initial context transformer of `append` is $\texttt{append}^{\alpha}(\Phi(\gamma; a, b, c)) = \mathrm{B}!\,\mathrm{B}!$. The demand $\Phi(\gamma; a, b, c)$ is first propagated to the branches of the <u>`case`</u>-expression.

$$\mathcal{E}^{\alpha}[\![\texttt{unfreeze y}]\!]\ \phi^{\alpha}\ (\Phi(\gamma; a, b, c)) \quad = \quad (xy{:}\ \mathrm{B}?\ \Phi(\gamma; a, b, c)!, \varepsilon{:}\ \texttt{ide})$$

The analysis of the second branch uses the initial context transformer.

$$\begin{aligned}
&\mathcal{E}^{\alpha}[\![\texttt{a:freeze (append w y)}]\!]\ \phi^{\alpha}\ (\Phi(\gamma; a, b, c))\\
=\ &\mathcal{E}^{\alpha}[\![\texttt{a (freeze (append w y))}]\!]\ \phi^{\alpha}\ (\gamma b\ \Phi(\gamma; a, b, c)c)\\
=\ &\mathcal{E}^{\alpha}[\![\texttt{a}]\!]\ \phi^{\alpha}\ (\gamma b)\ \&^{\alpha}\ \mathcal{E}^{\alpha}[\![\texttt{freeze (append w y)}]\!]\ \phi^{\alpha}\ (\Phi(\gamma; a, b, c)c)\\
=\ &\mathcal{E}^{\alpha}[\![\texttt{a}]\!]\ \phi^{\alpha}\ (\gamma b)\ \&^{\alpha}\ (c \rhd^{\alpha} \mathcal{E}^{\alpha}[\![\texttt{append w y}]\!]\ \phi^{\alpha}\ (\Phi(\gamma; a, b, c)c))\\
=\ &\mathcal{E}^{\alpha}[\![\texttt{a}]\!]\ \phi^{\alpha}\ (\gamma b)\ \&^{\alpha}\ (c \rhd^{\alpha} \mathcal{E}^{\alpha}[\![\texttt{w y}]\!]\ \phi^{\alpha}\ (\mathrm{B}!\,\mathrm{B}!))\\
=\ &(xy{:}\ \mathrm{B}?\,\mathrm{B}?, aw{:}\ \gamma b\,\mathrm{B}?)\ \&^{\alpha}\ (xy{:}\ \mathrm{B}c\,\mathrm{B}c, aw{:}\ \texttt{bot}c\,\mathrm{B}c)\\
=\ &(xy{:}\ \mathrm{B}c\,\mathrm{B}c, aw{:}\ \gamma(b \wedge c)\,\mathrm{B}c)
\end{aligned}$$

The discriminator `unfreeze x` is now analysed wrt $\mathrm{fold}[\mathrm{Nil}{:}\ \texttt{ide}]^{\alpha} = \Phi(\texttt{bot}; \texttt{ide}, !, !)$ and $\mathrm{fold}[\mathrm{Cons}{:}\ \gamma(b \wedge c)\,\mathrm{B}c]^{\alpha} = \Phi(\gamma; \texttt{bot}, b \wedge c, c)$.

$$\begin{aligned}
\mathcal{E}^{\alpha}[\![\texttt{unfreeze x}]\!]\ \phi^{\alpha}\ (\Phi(\texttt{bot}; \texttt{ide}, !, !)) &= (xy{:}\ \Phi(\texttt{bot}; \texttt{ide}, !, !)!\ \mathrm{B}?)\\
\mathcal{E}^{\alpha}[\![\texttt{unfreeze x}]\!]\ \phi^{\alpha}\ (\Phi(\gamma; \texttt{bot}, b \wedge c, c)) &= (xy{:}\ \Phi(\gamma; \texttt{bot}, b \wedge c, c)!\ \mathrm{B}?)
\end{aligned}$$

Combining the intermediate results we have

$$\begin{aligned}
&\mathcal{E}^{\alpha}[\![\underline{\texttt{case}}\ \texttt{unfreeze x}\ \ldots]\!]\ \phi^{\alpha}\ (\Phi(\gamma; a, b, c))\\
=\ &(xy{:}\ \mathrm{B}?\ \Phi(\gamma; a, b, c)!)\ \&^{\alpha}\ (xy{:}\ \Phi(\texttt{bot}; \texttt{ide}, !, !)!\ \mathrm{B}?) \sqcup^{\alpha}\\
&(xy{:}\ \mathrm{B}c\,\mathrm{B}c)\ \&^{\alpha}\ (xy{:}\ \Phi(\gamma; \texttt{bot}, b \wedge c, c)!\ \mathrm{B}?)\\
=\ &(xy{:}\ \Phi(\texttt{bot}; \texttt{ide}, !, !)!\ \Phi(\gamma; a, b, c)!) \sqcup^{\alpha} (xy{:}\ \Phi(\gamma; \texttt{bot}, b \wedge c, c)!\ \mathrm{B}c)\\
=\ &(xy{:}\ \Phi(\gamma; \texttt{ide}, b \wedge c, c)!\ \Phi(\gamma; a, b, c)c).
\end{aligned}$$

Hence, the first round yields $\texttt{append}^{\alpha}(\Phi(\gamma; a, b, c)) = \Phi(\gamma; \texttt{ide}, b \wedge c, c)!\ \Phi(\gamma; a, b, c)c$. The second round only differs in the result of the second branch which contains the recursive call to `append`.

$$\begin{aligned}
&\mathcal{E}^{\alpha}[\![\texttt{a:freeze (append w y)}]\!]\ \phi^{\alpha}\ (\Phi(\gamma; a, b, c))\\
=\ &\mathcal{E}^{\alpha}[\![\texttt{a}]\!]\ \phi^{\alpha}\ (\gamma b)\ \&^{\alpha}\ (c \rhd^{\alpha} \mathcal{E}^{\alpha}[\![\texttt{append w y}]\!]\ \phi^{\alpha}\ (\gamma b\ \Phi(\gamma; a, b, c)c))\\
=\ &\mathcal{E}^{\alpha}[\![\texttt{a}]\!]\ \phi^{\alpha}\ (\gamma b)\ \&^{\alpha}\ (c \rhd^{\alpha} \mathcal{E}^{\alpha}[\![\texttt{w y}]\!]\ \phi^{\alpha}\ (\Phi(\gamma; \texttt{ide}, b \wedge c, c)!\ \Phi(\gamma; a, b, c)c))\\
=\ &(xy{:}\ \mathrm{B}?\,\mathrm{B}?, aw{:}\ \gamma b\,\mathrm{B}?)\ \&^{\alpha}\\
&(xy{:}\ \mathrm{B}?\ \Phi(\gamma; a, b, c)c, aw{:}\ \texttt{bot}?\ \Phi(\gamma; \texttt{ide}, b \wedge c, c)!)\\
=\ &(xy{:}\ \mathrm{B}?\ \Phi(\gamma; a, b, c)c, aw{:}\ \gamma b\ \Phi(\gamma; \texttt{ide}, b \wedge c, c)!)
\end{aligned}$$

Analysing `unfreeze x` with respect to $\mathrm{fold}[\mathrm{Cons}{:}\ \gamma b\,\Phi(\gamma; \texttt{ide}, b \wedge c, c)!]^{\alpha} = \Phi(\gamma; \texttt{ide}, b, c)$ yields

$$\mathcal{E}^{\alpha}[\![\texttt{unfreeze x}]\!]\ \phi^{\alpha}\ (\Phi(\gamma; \texttt{ide}, b, c)) \quad = \quad (xy{:}\ \Phi(\gamma; \texttt{ide}, b, c)!\ \texttt{bot}?).$$

Putting the intermediate results together we get

$$
\begin{array}{rl}
& \mathcal{E}^{\alpha}[\![\underline{\texttt{case}}\ \texttt{unfreeze x} \ \ldots]\!]\ \phi^{\alpha}\ (\Phi(\gamma; a, b, c)) \\
= & (xy\colon \texttt{bot}?\ \Phi(\gamma; a, b, c)!)\ \&^{\alpha}\ (xy\colon \Phi(\texttt{bot}; \texttt{ide}, !, !)!\ \mathrm{B}?) \sqcup^{\alpha} \\
& (xy\colon \mathrm{B}?\ \Phi(\gamma; a, b, c)c)\ \&^{\alpha}\ (xy\colon \Phi(\gamma; \texttt{ide}, b, c)!\,\texttt{bot}?) \\
= & (xy\colon \Phi(\texttt{bot}; \texttt{ide}, !, !)!\ \Phi(\gamma; a, b, c)!) \sqcup^{\alpha} (xy\colon \Phi(\gamma; \texttt{bot}, b, c)!\ \mathrm{B}c) \\
= & (xy\colon \Phi(\gamma; \texttt{ide}, b, c)!\ \Phi(\gamma; a, b, c)c).
\end{array}
$$

Since the next round infers the same result we have found the least fixpoint. Hence the generic context transformer of append amounts to

$$\texttt{append}^{\alpha}(\mu\beta.\texttt{[]}\ a \mid \gamma b\colon\beta c) \quad = \quad (\mu\beta.\texttt{[]}\ \texttt{ide} \mid \gamma b\colon\beta c)!\ (\mu\beta.\texttt{[]}\ a \mid \gamma b\colon\beta c)c.$$

It is immediate from the equation above that append is lift strict in the first argument. Lift strictness in the second argument only holds if the demand on the result is tail strict. Furthermore, the generic context transformer makes explicit that the demand on the result propagates unchanged to the second argument and with a small change to the first argument (the empty list is acceptable as the first argument even if it is not acceptable as a result).

Idioms and shortcuts Before we proceed let us explain some often recurring Boolean terms. Assume that the demand on a function's result is $\mu\beta.\texttt{[]}\ a \mid \gamma b\colon\beta c$ and that the resulting demand on the argument has the lift $a \vee c$. Then the function is lift strict if $a \vee c$ evaluates to false which is only the case if the demand on the result is bottom. Analogously, $b \vee c$ specifies that lift strictness is given if the context is both head and tail strict. The idioms occurring in the sequel are listed below.

a: the empty list is not acceptable as a result ($a \cong \mathrm{P}$)

b: the demand is head strict ($b \cong \mathrm{H}$)

c: the demand is tail strict ($c \cong \mathrm{T}$)

$a \vee c$: the demand is bottom

$b \vee c$: the demand is head and tail strict

There is one further point which we would like to make. Due to the fact that not all assignments of truth values to Boolean variables must be considered ($\eta(\mu\beta.\texttt{[]}\ a \mid \gamma b\colon\beta c)$ with $\eta = \{a \mapsto \mathbf{f}, b \mapsto \mathbf{t}, c \mapsto \mathbf{f}\}$ does not yield a normal context and is hence negligible), different Boolean terms may, in fact, be regarded as equivalent. Take, for example, the term $ab \vee bc \equiv (a \vee c)b$ which evaluates to false if either both a and c are false or else b is false. Now, since b is necessarily false if both a and c are, $ab \vee bc$ may be simplified to b. [Hardware designers will probably identify valuations such as η above as don't care entries.] The following two simplifications are made subsequently (the terms are equivalent wrt the seven assignments which yield normal contexts).

$$
\begin{array}{rcl}
ab \vee c & \leadsto & b \vee c \\
ab \vee bc & \leadsto & b
\end{array}
$$

The same considerations apply to other principal contexts as well. The principal context on monomorphic lists, $\mu\beta.\texttt{[]}\ a \mid bc\colon\beta d$, gives rise to the following idioms.

a: the empty list is not acceptable as a result ($a \cong \mathrm{P}$)

b: the elements of the list are not required

c: the demand is head strict ($c \cong$ H)

d: the demand is tail strict ($d \cong$ T)

$a \vee d$: the demand is bottom

$c \vee d$: the demand is head and tail strict

$b \vee c$: only the empty list is acceptable as a result

Since only 11 out of 16 assignments yield normal contexts many Boolean terms may be identified. The rewrite rules listed below are applied to enhance the readability of the context transformers.

$$\begin{array}{rcl} ab \vee bd & \rightsquigarrow & b \\ bd \vee cd & \rightsquigarrow & d \\ ac \vee d & \rightsquigarrow & c \vee d \\ ac \vee bd \vee cd & \rightsquigarrow & c \vee d \\ b \vee c \vee d & \rightsquigarrow & b \vee c \\ ab \vee ac \vee bd \vee cd & \rightsquigarrow & b \vee c \end{array}$$

Standard list processing functions The context transformers listed below involve no approximations compared to the standard ones.

$$\begin{array}{lcl} \texttt{reverse}^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b{:}\beta c) & = & (\mu\beta.\texttt{[]}\ (a \vee c) \mid \gamma(b \vee c){:}\beta!)! \\ \texttt{sum}^\alpha(a) & = & (\mu\beta.\texttt{[]}\ a \mid a!{:}\beta!)! \\ \texttt{length}^\alpha(a) & = & (\mu\beta.\texttt{[]}\ a \mid \texttt{bot}a{:}\beta!)! \\ \texttt{and}^\alpha(\text{False}\ a \mid \text{True}\ b) & = & (\mu\beta.\texttt{[]}\ b \mid (\text{False}\ a \mid \text{True}\ \texttt{ide})!{:}\beta a)! \\ \texttt{concat}^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b{:}\beta c) & = & (\mu\beta.\texttt{[]}\ a \mid (\mu\delta.\texttt{[]}\ \texttt{ide} \mid \gamma b{:}\delta c)!{:}\beta c)! \\ \texttt{member}^\alpha(\text{False}\ a \mid \text{True}\ b) & = & (\mu\beta.\texttt{[]}\ a \mid \texttt{ide}!{:}\beta b)!\ (a \vee b)a \\ \texttt{take}^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b{:}\beta c) & = & (a \vee c)!\ (\mu\beta.\texttt{[]}\ a \mid \gamma b{:}\beta(a \vee c))a \\ \texttt{drop}^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b{:}\beta c) & = & \texttt{ide}!\ (\mu\beta.\texttt{[]}\ a \mid \gamma ?{:}\beta c)! \end{array}$$

The equations for the list transformers `reverse`, `take` and `drop` are worth studying. The argument of `reverse` is required head strict if the demand is head and tail strict. The function `take` preserves head strictness while its counterpart `drop` preserves tail strictness. On the negative side `take` does not propagate tail and `drop` does not propagate head strictness.

Sorting algorithms We have seen in Section 7.6.3 and 7.6.4 that different algorithmic solutions to the problem of sorting result in quite different context transformers (though the algorithms denote the same function which in turn implies that they possess the same least abstraction). Here are the generic context transformers.

$$\begin{array}{lcl} \texttt{sort}^\alpha(\mu\beta.\texttt{[]}\ a \mid bc{:}\beta d) & = & \text{T}\,(b \vee c)! \\ \texttt{msort}^\alpha(\mu\beta.\texttt{[]}\ a \mid bc{:}\beta d) & = & (\mu\beta.\texttt{[]}\ \texttt{ide} \mid (b \vee c)c{:}\beta!)! \\ \texttt{qsort}^\alpha(\mu\beta.\texttt{[]}\ a \mid bc{:}\beta d) & = & (\mu\beta.\texttt{[]}\ (a \vee d) \mid (b \vee c)(c \vee d){:}\beta!)! \\ \texttt{hsort}^\alpha(\mu\beta.\texttt{[]}\ a \mid bc{:}\beta d) & = & (\mu\beta.\texttt{[]}\ \texttt{ide} \mid (b \vee c)c{:}\beta!)! \\ \texttt{bhsort}^\alpha(\mu\beta.\texttt{[]}\ a \mid bc{:}\beta d) & = & \text{T}\,(b \vee c)! \end{array}$$

Note that the transformers with the exception of $\texttt{qsort}^\alpha$ involve minor approximations as they do not propagate the bottom context (all Boolean variables set to false). Note, too, that the principal generic context on the left hand side involves four Boolean variables since the sorting functions are of type $\texttt{[num]} \rightarrow \texttt{[num]}$.

All sorting algorithms are tail strict, bottom-up mergesort and top-down heapsort are head strict if the demand is head strict, quicksort is head strict if the demand is head and tail strict, and top-down mergesort and bottom-up heapsort are never head strict. The elements of the list are not required in case that only `[]` is acceptable as a result.

Primes The context transformer of `primes` shows that strictness analysis can to a certain extent also infer termination properties. The generic transformer comprises the same information, that is, `primes` does not terminate if the demand is tail strict.

$$\texttt{primes}^\alpha(\mu\beta.\texttt{[]}\ a \mid bc\!:\!\beta d) \quad = \quad d$$

8.6.3 Functions on trees

`size` Since `size` plays the same rôle as `length` does for lists we obtain a similar generic context transformer: The labels are not needed while the recursive components are required.

$$\texttt{size}^\alpha(a) \quad = \quad (\mu\beta.\text{Empty}\ a \mid \text{Node}\ \beta!\,\texttt{bot}a\,\beta!)!$$

Tree traversal We have seen in Section 7.6.4 that different variants of inorder tree traversal possess the same context transformer which demonstrates that the analysis is to a certain extent insensitive to variations in the coding of algorithms. This essentially also holds in the generic case.

$$\begin{aligned}
\texttt{inorder}_1^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b\!:\!\beta c) &= (\mu\beta.\text{Empty}\ \texttt{ide} \mid \text{Node}\ \beta!\,\gamma(b \vee c)\,\beta c)! \\
\texttt{inorder}_2^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b\!:\!\beta c) &= (\mu\beta.\text{Empty}\ (a \vee c) \mid \text{Node}\ \beta!\,\gamma(b \vee c)\,\beta c)! \\
\texttt{inorder}_3^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b\!:\!\beta c) &= (\mu\beta.\text{Empty}\ \texttt{ide} \mid \text{Node}\ \beta!\,\gamma(b \vee c)\,\beta c)!
\end{aligned}$$

Note that the first and the third context transformer do not preserve the bottom context while the second one does. The context transformer for the three variants of `preorder` exhibit a similar behaviour.

$$\begin{aligned}
\texttt{preorder}_1^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b\!:\!\beta c) &= (\mu\beta.\text{Empty}\ \texttt{ide} \mid \text{Node}\ \beta c\,\gamma b\,\beta c)! \\
\texttt{preorder}_2^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b\!:\!\beta c) &= (\mu\beta.\text{Empty}\ (a \vee c) \mid \text{Node}\ \beta c\,\gamma b\,\beta c)! \\
\texttt{preorder}_3^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b\!:\!\beta c) &= (\mu\beta.\text{Empty}\ \texttt{ide} \mid \text{Node}\ \beta c\,\gamma b\,\beta c)!
\end{aligned}$$

All six transformer show very clearly how head and tail strict demands on the results are mapped to demands on the functions' arguments. The transformer of the auxiliary function `flatten` is equally easy to read.

$$\texttt{flatten}^\alpha(\mu\beta.\texttt{[]}\ a \mid \gamma b\!:\!\beta c) \quad = \quad (\mu\beta.\texttt{[]}\ a \mid (\mu\delta.\text{Empty}\ \texttt{ide} \mid \text{Node}\ \delta c\,\gamma b\,\delta c)!\!:\!\beta c)!$$

8.6.4 A propositional theorem prover

The theorem prover for propositional logic described in Section B.5 is the first more substantial program we are going to analyse. Recall that we did not consider this example in Section 7.6.5 since the context transformers involved are too large to be manageable. However, using the technique presented in this chapter the analysis becomes feasible.

The tautology checker The checker which is based on the sequent calculus proceeds as follows: Starting with the sequent $\vdash\ \phi$ a deduction tree is build by repeatedly applying the inference rules of the sequent calculus. It is important to note that the formulae contained in the sequents are considered from left to right. The deduction tree is then traversed in a depth-first fashion returning a list of sequents labelled to the leaves. By deleting axioms from this list a list of counterexamples is obtained. The given formula ϕ is valid iff the list of counterexamples is empty. Lazy evaluation causes the search to stop if the first counterexample is found. Despite the apparent complexity of the program the inferred context transformer admirably mirrors the operational behaviour of the tautology checker.

$$\begin{aligned}&\texttt{taut}^{\alpha}(\text{False } a \mid \text{True } b)\\ =\ &(\mu\beta.\text{Atom } (a \vee b)a \mid \text{Not } \beta! \mid \text{And } \beta!\, \beta a \mid \text{Or } \beta!\, \beta a \mid \text{Then } \beta!\, \beta a \mid \text{Iff } \beta!\, \beta!)!\end{aligned}$$

If both `False` and `True` are acceptable as results then `taut` is lift strict and left strict, that is, the left subformula of each connective is required recursively. In the case of `Iff` even both subformulas are needed. It is amazing that the demand on each connective precisely mirrors the behaviour of its semantic counterpart: Negation is strict, conjunction, disjunction, and implication are strict in their first arguments while equivalence is bistrict.

Now, if only `True` is acceptable as a result then the complete formula is required! This property is due to the construction of so-called atomically closed deduction trees, that is, trees in which every sequent labelled to a leaf is atomic. [Things would be different if we modified the function `finished` so that the expansion stops if an (not necessarily atomic) axiom is found: `taut` $(a \vee \neg a \vee \bot) \Rightarrow$ `True`.]

The situation changes if the deduction tree is traversed breadth-first. In this case a much weaker transformer is inferred.

$$\begin{aligned}&\texttt{taut}^{\alpha}(\text{False } a \mid \text{True } b)\\ =\ &(\mu\beta.\text{Atom } \texttt{ide} a \mid \text{Not } \beta a \mid \text{And } \beta a\, \beta a \mid \text{Or } \beta a\, \beta a \mid \text{Then } \beta a\, \beta a \mid \text{Iff } \beta a\, \beta a)a\end{aligned}$$

If both `False` and `True` are acceptable as results then `taut` is completely lazy (well, this involves an approximation as `taut` should be at least lift strict). On the other hand, if we insist on getting a positive result then `taut` becomes completely strict again.

Disjunctive normalform The function `dnf` converts a given formula to an equivalent proposition in disjunctive normalform. Its context transformer resembles in many respects the one for `taut`.

$$\begin{aligned}&\texttt{dnf}^{\alpha}(\mu\beta.\text{Atom } (a)b \mid \text{Not } \beta c \mid \text{And } \beta d\, \beta e \mid \text{Or } \beta f\, \beta g \mid \text{Then } \beta h\, \beta i \mid \text{Iff } \beta j\, \beta k)\\ =\ &(\mu\beta.\text{Atom } \texttt{ide}(b \vee c \vee d \vee e \vee f \vee g) \mid\\ &\qquad \text{Not } \beta! \mid \text{And } \beta!\, \beta g \mid \text{Or } \beta!\, \beta g \mid \text{Then } \beta!\, \beta g \mid \text{Iff } \beta!\, \beta!)!\end{aligned}$$

Thus `dnf` is strict and lift strict. The demand on atoms mirrors the fact that the disjunctive normalform only contains the connectives $\neg$, $\wedge$, and $\vee$. The disjunctive normalform is of the form `Or` ϕ_1 `(Or` $\phi_2 \cdots$ `(Or` $\phi_{n+1}\ \phi_n$`)`$\cdots$`)`, that is, the constructor `Or` is used in a list-like fashion. Now, if the result is needed 'tail strict wrt `Or`' then the argument is required in toto.

Changing the search strategy to breadth-first weakens again the inferred context transformer.

$$\begin{aligned}&\texttt{dnf}^{\alpha}(\mu\beta.\text{Atom } (a)b \mid \text{Not } \beta c \mid \text{And } \beta d\, \beta e \mid \text{Or } \beta f\, \beta g \mid \text{Then } \beta h\, \beta i \mid \text{Iff } \beta j\, \beta k)\\ =\ &(\mu\beta.\text{Atom } \texttt{ide}(b \vee c \vee d \vee e \vee f \vee g) \mid\\ &\qquad \text{Not } \beta g \mid \text{And } \beta g\, \beta g \mid \text{Or } \beta g\, \beta g \mid \text{Then } \beta g\, \beta g \mid \text{Iff } \beta g\, \beta g)g\end{aligned}$$

In this case `dnf` is either completely lazy or completely strict depending on the lift of `Or`'s second argument.

8.6.5 A pretty-printer for first-order expressions

As the final example we consider the pretty-printer introduced in Section B.6. The presentation of the results profits very much from the compact representation which establishes clearly the connection between the demands on the result and the resulting demands on the argument.

As usual we discuss only the main functions. The conversion of a first-order expression to a string proceeds in two steps. First, an expression is mapped to an element of the data type `text` which is essentially a nested piece of text incorporating information about indentation and line breaking. The text which reflects the logical structure of the expression is then mapped to a string by a relatively simple tree traversal. The function `display` carries out the second step, its context transformer is given below.

$$\begin{array}{lcl} \texttt{display}^{\alpha}(\mu\beta.\texttt{[]}\ a \mid bc\!:\!\beta d) & = & (\mu\beta.\mathrm{Str}\ (\mu\delta.\texttt{[]}\ (a \vee d) \mid b(c \vee d)\!:\!\delta d)! \mid \\ & & \quad \mathrm{Block}\ (\mu\delta.\texttt{[]}\ (a \vee d) \mid \beta!\!:\!\delta d)! \mid \\ & & \quad \mathrm{Indent}\ \beta! \\ & & \quad \mathrm{NL}\ (b \vee c)! \end{array}$$

First thing to note is that the different constructors are lift strict and that `Block` is additionally head strict (so this is the part of the argument which must be inspected before a single character is put on the output list). Since `NL` also produces text, `Str` is head strict only if the demand is both head and tail strict. In this case the complete argument is evaluated. Finally, note that `NL` is not acceptable as an argument if only the empty list is acceptable as a result.

Before explaining the context transformer of `showexpr` it is advisable to introduce some abbreviations. Recall that the type of `showexpr` is `expr [char]` $\rightarrow$ `[char]` which gives rise to a deeply nested type structure. The constructor `Case`, for example, takes a list of triples which contain a list of strings in the second component.

$$\begin{array}{lcl} \kappa_1 & = & \mu\delta.\texttt{[]}\ (b \vee c) \mid b(c \vee d)\!:\!\delta d \\ \kappa_2 & = & \mu\delta.\texttt{[]}\ (b \vee c) \mid \kappa_1 d\!:\!\delta d \end{array}$$

The context κ_1 on strings essentially corresponds to the one on `Str` above, κ_2 maps κ_1 to lists of strings. Using these two abbreviations the context transformer for `showexpr` amounts to

$$\begin{array}{l} \texttt{showexpr}^{\alpha}(\mu\beta.\texttt{[]}\ a \mid bc\!:\!\beta d) \\ = \ (\mu\beta.\mathrm{Var}\ (\delta.\texttt{[]}\ (a \vee d) \mid b(c \vee d)\!:\!\beta d)! \mid \\ \qquad \mathrm{Freeze}\ \beta d \mid \\ \qquad \mathrm{Unfreeze}\ \beta d \mid \\ \qquad \mathrm{Sys}\ \kappa_1 d\ \beta d \mid \\ \qquad \mathrm{Con}\ \kappa_1 d\ \beta d \mid \\ \qquad \mathrm{Fun}\ \kappa_1 d\ \beta d \mid \\ \qquad \mathrm{Sequ}\ (\mu\delta.\texttt{[]}\ (a \vee d) \mid \beta d\!:\!\delta d)! \mid \\ \qquad \mathrm{Case}\ \beta d\ (\mu\delta.\texttt{[]}\ (b \vee c) \mid (\kappa_1 d, \kappa_2 d, \beta d)!\!:\!\delta d)d \mid \\ \qquad \mathrm{Let}\ \kappa_2 d\ \beta d\ \beta d \mid \\ \qquad \mathrm{Rec}\ \kappa_2 d\ \beta d)!. \end{array}$$

Thus if the demand on the result is tail strict then the complete structure of the expression may be evaluated in advance. If the demand is additionally head strict then the strings embedded in the expression are required as well. It is interesting to note that most constructors are not acceptable as arguments if only the empty list may be returned by `showexpr`. This is due to the output of keywords such as `Freeze`, `Case` etc.

8.7 Run-time complexity

At first sight the technique presented in this chapter appears to improve dramatically on the algorithm of Chapter 7. While this certainly holds for the average-case behaviour (see the benchmarks at the end of this section) it does not prove true with respect to the worst-case behaviour.

We have certainly improved on the number of principal (generic) contexts. The naïve implementation has to consider approximately 2^n principal contexts n being the size of the result type whereas the generic analysis considers only a single one! Theoretically, this gain is compensated by the additional complexity of the primitive operations ($\sqcup^\alpha$, $\&^\alpha$ etc). Recall that disjunction and conjunction of contexts are finally mapped to disjunction and conjunction of minimal polynoms. So let us first work out the complexity of these operations. A minimal polynom may have size exponential in the number of Boolean variables which is bounded by n. The worst-case is exhibited by the majority function which has $\binom{n}{\lceil n/2 \rceil} = \mathrm{O}(2^n/\sqrt{n})$ conjuncts each containing $\lceil n/2 \rceil$ Boolean variables. Let $|p|$ be the size of the minimal polynom p. The cost of the disjunction $p \veebar q$ is given by $\mathrm{O}(|p| \cdot |q|)$. Accordingly, the cost of the conjunction $p \barwedge q$ amounts to $\mathrm{O}(|p|^2 \cdot |q|^2)$.

Now, minimal polynoms are certainly not the best choice for representing and manipulating (monotone) Boolean functions. Alternatives include Ordered Binary-Decision Diagrams (OBDDs) [27] which are wildly used in digital system design and Typed Decision Graph (TDG) [114] being a refinement of OBDDs. Interestingly, the last paper proposes to use TDGs in abstract interpretation (ideal-based forward analysis) reporting a great increase in efficiency. However, with respect to the worst-case complexity both representation do not improve significantly on minimal polynoms: Both may have size exponential in the number of variables. Furthermore, the cost of conjunction and disjunction is $\mathrm{O}(|p| \cdot |q|)$ where $|p|$ is the size of the OBDD respectively of the TDG representing p [27, 114].

Turning to the complexity of the generic analysis we first note that the length of the fixpoint iteration is the same for both analyses since the generic analysis faithfully simulates the standard one. Assume that the function $f :: \sigma_1 \to \sigma_2$ is analysed. Let s be the size of f's definition, let $m = \mathrm{size}(\sigma_1)$ and $n = \mathrm{size}(\sigma_2)$. Using an OBDD representation of Boolean functions the complexity is made up of

1. length of fixpoint iteration: $\mathrm{O}(m \cdot 2^n)$,
2. number of principal generic contexts: $\mathrm{O}(1)$,
3. traversal of the expression tree: $\mathrm{O}(s)$,
4. cost of primitive operations: $\mathrm{O}(s \cdot 4^n)$,

leading to a worst-case complexity of $\mathrm{O}(m \cdot 8^n \cdot s^2)$ which is actually *worse* than the complexity of the standard analysis (cf Section 7.7). Fortunately, it is not difficult to improve the implementation so as to obtain a more satisfactory result.

We have seen that the operations on contexts may be reduced to operations on monotone Boolean functions. Therefore it stands to reason to shift also the fixpoint iteration from the realm of (generic) context transformers to the realm of monotone Boolean functions. To give an impression how this idea may be put to work consider the context transformer of `append`.

$$
\begin{aligned}
&\mathtt{append}^\alpha(\mu\beta.\mathtt{[]}\ a \mid \gamma b \mathbin{:} \beta c) \\
=\ &(\mu\beta.\mathtt{[]}\ \Phi_1(a,b,c) \mid (\gamma \wedge \Phi_2(a,b,c))\Phi_3(a,b,c) \mathbin{:} \beta\Phi_4(a,b,c))\Phi_5(a,b,c) \\
&(\mu\beta.\mathtt{[]}\ \Phi_6(a,b,c) \mid (\gamma \wedge \Phi_7(a,b,c))\Phi_8(a,b,c) \mathbin{:} \beta\Phi_9(a,b,c))\Phi_{10}(a,b,c)
\end{aligned}
$$

The ten ternary Boolean functions are defined by

$$\begin{array}{rclcrcl}
\Phi_1(a,b,c) & = & 1 & \quad & \Phi_6(a,b,c) & = & a \\
\Phi_2(a,b,c) & = & 1 & & \Phi_7(a,b,c) & = & 1 \\
\Phi_3(a,b,c) & = & b & & \Phi_8(a,b,c) & = & b \\
\Phi_4(a,b,c) & = & c & & \Phi_9(a,b,c) & = & c \\
\Phi_5(a,b,c) & = & 0 & & \Phi_{10}(a,b,c) & = & c.
\end{array}$$

The context transformer is 'usually' calculated as follows: First the Φ_i are set to $\Phi_i^0(a,b,c) = 0$, the analysis is performed obtaining new approximations Φ_i^1, the analysis is repeated with Φ_i^1 obtaining new approximations Φ_i^2 and so forth until $\Phi_i^k = \Phi_i^{k+1}$, for all i. Now, instead of repeating the analysis several times we carry it out only once using the above context transformer where the $\Phi_i(a,b,c)$ are interpreted symbolically! Of course, the set of Boolean terms must be extended as to include terms of the form $\Phi_i(a,b,c)$. From the resulting context transformer we may then extract a set of fixpoint equations defining the Φ_i. For `append` we obtain the following, very simple system

$$\begin{array}{rclcrcl}
\Phi_1(a,b,c) & = & 1 & \quad & \Phi_6(a,b,c) & = & a \vee \Phi_6(a,b,c) \\
\Phi_2(a,b,c) & = & 1 & & \Phi_7(a,b,c) & = & 1 \\
\Phi_3(a,b,c) & = & \Phi_3(a,b,c) \vee b & & \Phi_8(a,b,c) & = & b \vee \Phi_8(a,b,c) \\
\Phi_4(a,b,c) & = & \Phi_4(a,b,c) \vee \Phi_5(a,b,c)c & & \Phi_9(a,b,c) & = & c \vee \Phi_9(a,b,c) \\
\Phi_5(a,b,c) & = & 0 & & \Phi_{10}(a,b,c) & = & c \vee \Phi_{10}(a,b,c),
\end{array}$$

which has the solution shown above.

The sketched reduction to fixpoint equations has several advantages: The set of equations can be sorted into minimal mutually recursive groups speeding up the calculation of the fixpoint. Such a dependency analysis is usually applied on the level of function definitions. Here, a much finer level of granularity is obtained. The most important advantage is certainly that fixpoint equations in the domain of monotone Boolean functions have been in use for fifteen years in first-order forward analysis! So we may profit from the wealth of representation and methods which were developed over the years: frontiers [35], λ-expressions [34], TDGs [114], symbolic fixpoint iteration [36], neededness-based chaotic iteration [98].

Turning to the complexity of this method first note that evaluating a recursively defined monotone Boolean function at any point is complete in deterministic exponential time in the length of the function definition [76].

The computation of the equation system takes $\mathrm{O}(s^2)$ steps in the worst-case, since the expression tree is traversed once and the cost of a primitive operation is at most proportional to the size of f. The size of the resulting equation system is bounded by $\mathrm{O}(s^2)$, as well. The most expensive operation is, of course, the computation of the least solution to that system. Using truth tables for representing elements of $\mathbb{B}^m \to \mathbb{B}^n$ we have

1. length of fixpoint iteration: $\mathrm{O}(m \cdot 2^n)$,
2. number of entries in the truth table: $\mathrm{O}(2^n)$,
3. computation of an entry: $\mathrm{O}(s^2)$,

summing up to a worst-case behaviour of $\mathrm{O}(m \cdot 4^n \cdot s^2)$ which coincides with the complexity of the standard analysis. However, we do not expect that the analysis of a user-defined function exhibits such a bad behaviour. The practical experiments discussed below strongly support this conjecture.

	standard analysis		generic analysis		generic (only lifts)	
function	cpu[sec]	#fpi	cpu[sec]	Δcpu[%]	cpu[sec]	Δcpu[%]
`sort`	17.4	4	3.8	460	3.0	580
`msort`	67.8	4	4.3	1580	2.8	2420
`qsort`	1287.4	4	5.2	24760	2.8	45980
`hsort`	39.9	4	5.9	680	4.0	1000
`bhsort`	5921.4	4	19.7	30060	9.0	65790
theorem prover	—†	7	338.0	—	94.9	—
pretty printer	—†	7	2899.8	—	77.5	—
†not enough heap space (2.500.000 cells)						

Table 8.5: Running times of standard and generic analysis

Let us conclude the section with some benchmarks. The MIRANDA implementation of the generic analysis serves as the basis for the comparative study listed in Table 8.5. Times are given in seconds (user and system time) on a Sun SparcStation 10 with 48 Mbytes of memory. The running time of the standard analysis is listed in the first column.[3] Note that the times include parsing of the source code and printing the results. The second column contains the number of fixpoint iterations which are surprisingly small compared to the theoretical upper bound of $O(m \cdot 2^n)$. The third column lists the respective running times of the generic analysis; the relative improvement in percent is given in the fourth column.

The results are quite striking. The analysis of `qsort`, for example, is about 250 times faster. The bad performance of the standard analysis is, of course, due to the size of the context transformers which are involved in the analysis. The auxiliary function `partition` employed in the definition of `qsort` comprises, for instance, 442 entries. Even more impressive `buildn` which is used by `bhsort` contains 694 entries. The analysis of the theorem prover and the pretty printer is even infeasible (cf Section 7.6.5 and 7.6.6 respectively).

The greater complexity of the last two programs compared to the sorting routines is also reflected in the running times of the generic analysis. However, the absolute times should be considered with care. The MIRANDA interpreter, which is fun to use, certainly ranks among the slowest systems. Using a native code compiler is likely to give a significant speedup.

Speedup may be also gained by giving up certain information. The last two columns in Table 8.5 give the results for a variant of the generic analysis which takes only lifts into account (on system-defined types only `ide` is considered). Little is lost by this restriction since the additional information can seldom be put to use by the code generator. The speedup is, however, quite impressing. The pretty printer is analysed about 40 times faster.

[3]The standard analysis is realized as a special case of the generic analysis. Consequently, some primitive operations involve a small run-time overhead.

Chapter 9

Conclusion

There are two things which I am confident I can do very well: one is an introduction to any literary work, stating what it is to contain, and how it should be executed in the most perfect manner; the other is a conclusion, shewing from various causes why the execution has not been equal to what the author promised to himself and to the public.

— Samuel Johnson, *Boswell Life vol. 1, p. 291 (1755)*

This chapter summarizes the main results and holds out a prospect of future work.

9.1 Summary

In Chapter 2 we introduced the syntax of a first-order, polymorphic, non-strict language with algebraic data types. In order to simplify the presentation of the various semantics a number of transformations were applied to the language: The call-by-value transformation yields a strict language containing unevaluated expressions as first-class objects. Semantically, the transformation is justified by the domain isomorphism mapping $D \to E$ onto $D_\perp \circ\!\!\to E$. Working in the latter space makes is possible to speak about the necessity of information. The de Bruijn transformation concretizes environments as nested tuples and variables as projection functions.

In Chapter 3 we gave a denotational semantics for the first-order language respectively its strict, lifted de Bruijn variant.

In Chapter 4 a theory of projections was developed collecting and extending results from various sources, most notably [150, 64, 47]. We characterized in particular projections on lifts, coalesced sums, and smash products thereby giving operations to synthesize new projections and to analyse given projections. Projections on smash products were classified either as independent or as dependent. Projections of the former type can be represented by tuples of projections (each projection working on a component domain). However, independent projections are only able to specify isolated strictness. To express joint strictness one must resort to dependent projections which may be represented by sets of independent projections.

In Chapter 5 we proved that every continuous function whose domain satisfies Axiom I possesses a least abstraction which in turn characterizes the function—provided it is strict. This result corrects a misconception of Davis [46], namely that the 'parallel or' does not have a least abstraction. However, due to the presence of parallel functions least abstractions fail to be compositional. We showed that this desirable property is restored if one focuses on stable functions of Berry. Building on the theory of stable functions we then developed a non-standard semantics for the first-order language presented in Chapter 2. The non-standard semantics maps a textual function to the least abstraction of its standard denotation. This improves on previous work: The abstract semantics given, for example, in [150] or [105] yields only a safe approximation of the least abstraction.

In Chapter 6 we introduced finite abstract domains for every type, so-called context lattices, which approximate the corresponding, generally infinite projection lattices. We focused in particular on the representation of projections on polymorphic types extending previous work on this subject [81]. It is interesting to note that our approach has a very algebraic flavour contrasting to the category-theoretic approach of Hughes in *loc. cit.* Finally, abstract variants of various operators on projections were defined and proved safe.

In Chapter 7 we specified an approximation semantics and proved it safe with respect to the non-standard semantics of Chapter 5. The rules given refine those of [105]. Due to the use of polymorphic contexts the analysis is 'nearly' polymorphically invariant (only in the case of bottom contexts approximations are introduced). The analysis was then put on the test bench by determining the abstractions of numerous list- and tree-processing functions. The inferred results nicely demonstrated the analytical power of projection-based backward analysis. However, we also pointed out shortcomings of the technique being due to the exponential growth of context lattices.

In Chapter 8 we addressed the aforementioned problem of combinatorial explosion. Building on a compact representation of abstractions a new method, termed generic analysis, with a dramatically improved average-case behaviour was devised. Generic analysis may be viewed as a reduction of projection-based backward analysis to the problem of finding fixpoints in the domain of monotone Boolean functions. This is especially of practical interest since a number of good algorithms have been developed for the latter problem. Finally, the new method was applied to two non-trivial programs yielding good results in a reasonable amount of time.

9.2 Future work

Higher-order analysis It remains to extend the analysis to languages with higher-order functions. As matters stand this extension is far from being trivial. It is in particular the combination of higher-order functions and polymorphism which causes problems.

For that reason let us first consider monomorphic higher-order functions. Previous approaches either involve a combination of forward and backward analysis [87, 50], higher-order arguments making up the forward component, or are based on a relational analysis [33]. Obviously, both approaches are difficult to implement due to the additional level of complexity. However, we strongly believe that the generic analysis may attribute to the design of a good average-case algorithm. Let us illustrate the claim by 'analysing' the functions `map` and `foldr`.

```
map :: (α1 → α2) [α1] → [α2]
map f x = case x of [] → [] |
                    a:w → f a:map f w

foldr :: (α1 α2 → α2) α2 [α1] → α2
foldr (⊕) e x = case x of [] → e |
                          a:w → a ⊕ foldr (⊕) e w
```

Assume for the moment that $\alpha_1 = \alpha_2 = \mathtt{num}$. In the higher-order case the evaluation of arguments does not only depend on the surrounding context but on the strictness properties of functional arguments as well. The second argument of `map`, for example, is required head strict if the context is head strict *and* the first argument is a lift strict function. Even more typical the evaluation of the third argument of `foldr` depends only on its first argument. The behaviour of the two functions may be summarized as follows (note that we only consider lifts).

$$\begin{array}{lcl} \mathtt{map}^{\alpha}(\mu\beta.\,\mathtt{[]} \mid \mathtt{num}a\,{:}\,\beta b) & = & (\mathtt{num}c \rightarrow \mathtt{num})?\,(\mu\beta.\,\mathtt{[]} \mid \mathtt{num}(a \vee c)\,{:}\,\beta b)! \\ \mathtt{foldr}^{\alpha}(\mathtt{num}) & = & (\mathtt{num}a\,\mathtt{num}b \rightarrow \mathtt{num})?\,\mathtt{num}b\,(\mu\beta.\,\mathtt{[]} \mid \mathtt{num}a\,{:}\,\beta b)! \end{array}$$

Contrary to the first-order case the abstractions of `map` and `foldr` contain free Boolean variables playing the rôle of placeholders. The subterms $\mathtt{num}c \rightarrow \mathtt{num}$ and $\mathtt{num}a\,\mathtt{num}b \rightarrow \mathtt{num}$ may be considered as templates for concrete abstractions. If `foldr` is applied to a function, say, `f` the template is instantiated to the abstraction of `f`. Assume for the sake of example that `f` is only lift strict in the second argument, then a is bound to '?' and b to '!'. Consequently, `foldr f` is lift strict in both arguments and additionally tail strict in the second. Thus Boolean variables allow to express interdependence of arguments.

The specification of a template for a context transformer poses no problems in the monomorphic case, because the structure of the respective function is known in advance. This no longer holds in the polymorphic case. The type $\alpha_1\,\alpha_2 \rightarrow \alpha_2$ of `foldr`'s first argument does not reveal anything about the infrastructure of the function. This seems to be a serious problem, since `foldr` may be applied to functions as different as `+`, `&`, `:`, `append` etc. The question arises as to whether there is a uniform way of describing the behaviour of `foldr`. Recall that we were faced with a similar problem in the analysis of first-order functions. To this end we invented context variables and symbolic operations on them. There is hope that context variables may also attribute to the analysis of higher-order functions. Here are the context transformers of `map` and `foldr`.

$$\begin{array}{lcl} \mathtt{map}^{\alpha}(\mu\beta.\,\mathtt{[]} \mid \gamma_2 a\,{:}\,\beta b) & = & (\gamma_1 c \rightarrow \gamma_2)?\,(\mu\beta.\,\mathtt{[]} \mid \gamma_1(a \vee c)\,{:}\,\beta b)! \\ \mathtt{foldr}^{\alpha}(\gamma_2) & = & (\gamma_1 a\,\gamma_2' b \rightarrow \gamma_2)?\,\gamma_2' b\,(\mu\beta.\,\mathtt{[]} \mid \gamma_1 a\,{:}\,\beta b)! \end{array}$$

The subterms $\gamma_1 c \rightarrow \gamma_2$ and $\gamma_1 a\,\gamma_2' b \rightarrow \gamma_2$ serve again as templates for concrete abstractions. In Section 7.6.3 we have defined a number of functions in terms of `foldr`. It is interesting to note that these definitions yield the same context transformers as the respective first-order ones (if only lifts are considered). As a final example consider the following definitions of `append` and `concat`.

```
append x y = foldr (:) y x
concat = foldr append []
```

Instantiating $\gamma_1 a\,\gamma_2' b \rightarrow \gamma_2$ to the context transformers of '`:`' and `append` we obtain

$$\begin{array}{lcl} \mathtt{append}^{\alpha}(\mu\beta.\,\mathtt{[]} \mid \gamma a\,{:}\,\beta b) & = & (\mu\beta.\,\mathtt{[]} \mid \gamma a\,{:}\,\beta b)!\,(\mu\beta.\,\mathtt{[]} \mid \gamma a\,{:}\,\beta b)?, \\ \mathtt{concat}^{\alpha}(\mu\beta.\,\mathtt{[]} \mid \gamma a\,{:}\,\beta b) & = & (\mu\beta.\,\mathtt{[]} \mid (\mu\delta.\,\mathtt{[]} \mid \gamma a\,{:}\,\delta b)!\,{:}\,\beta b)!. \end{array}$$

A prototypical implementation building on the above representation of context transformers was developed by Michael Kettler [102]. However, the presentation is in parts speculative and much remains to be done to provide a sound formal basis for the technique.

Code generation The primary motivation for strictness analysis is to infer information for improving the generation of code. We hinted at possible optimizations in the introduction to this dissertation. Nonetheless, the problem of putting strictness information to use is far from being settled. Surprisingly, this is a very neglected topic: Compared to the huge amount of work on strictness analysis there is little work on this subject, but see [65, 25, 31, 125, 44, 58, 129, 74].

We have seen that the behaviour of a function strongly depends on its invocation context. Hence, one solution is to create specialized versions of a function for each actual context in which the function is called. If care is taken as to avoid the trap of increased space complexity (cf Section 1.2) this approach is likely to give more efficient code. However, the author knows of no framework which allows to formally justify the claim of improved code quality. An obvious disadvantage of the approach is that the compiled program may become very large or even unwieldy. In a sense the problem of combinatorial explosion re-enters the stage. The problem is especially challenging in connection with separate compilation.

Appendix A

Order and Domain Theory

Mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.
— Bertrand Russell, *Mysticism and Logic (1918) ch. 4*

Angling may be said to be so like the mathematics, that it can never be fully learnt.
— Izaak Walton, *The Compleat Angler (1653)*

This appendix reviews elementary material about ordered sets and domains. Its main purpose is to fix notation and terminology and to provide a basis for the more mathematically oriented chapters. As a consequence the presentation is rather terse and unsuited for an introduction to these topics. For a more comprehensive account we refer to the excellent textbook by Davey and Priestley [45] which largely covers Section A.1 up to A.6. Complementary material on the application of order theory to denotational semantics is to be found in [142, 116, 143].

Plan of the chapter Section A.1 introduces basic facts about ordered sets. Semilattices and lattices are treated in Section A.2. Complete partial orders and fixpoints of continuous functions are studied in Section A.3. Section A.4 deals with various ways of building new cpos. Complete semilattices and lattices, which are used intensively throughout this work, are subject of Section A.5. Section A.6 is concerned with algebraicity which is central to the representation theory of Chapter 4. Section A.7 presents basic material about projections and demonstrates their use in solving recursive domain equations. Finally, Section A.8 introduces I-domains and stable functions.

A.1 Ordered sets

A *pre-order* (or *quasi-order*) on a set P is a binary relation $\preccurlyeq$ which is reflexive and transitive. A *partial order* on a set P is a binary relation $\sqsubseteq$ which is reflexive, anti-symmetric and transitive. A set P equipped with an order relation is said to be a *partially ordered set* (notation: $\langle P; \sqsubseteq\rangle$). A pre-order $\preccurlyeq$ on P induces an equivalence relation $x \equiv y \pmod{\theta} :\Longleftrightarrow x \preccurlyeq y \wedge y \preccurlyeq x$ and a partial order on the quotient $\langle P/\theta; \sqsubseteq\rangle$ with $[x]_\theta \sqsubseteq [y]_\theta :\Longleftrightarrow x \preccurlyeq y$. We will usually abuse notation and use the same symbol for both $\preccurlyeq$ and $\sqsubseteq$.

The dual of a partially ordered set P (notation: P^δ) is obtained by reversing the order: $x \sqsubseteq_{P^\delta} y$ if and only if $y \sqsubseteq_P x$. An order relation $\sqsubseteq$ gives rise to a *strict order*: $x \sqsubset y$ if and only if $x \sqsubseteq y$ and $x \neq y$. An element x is *covered by* y (notation: $x \prec y$) if and only if $x \sqsubset y$ and $(\forall z \in P)\ x \sqsubseteq z \sqsubset y \Longrightarrow x = z$.

Finite ordered sets may be drawn using so-called *Hasse diagrams* which essentially depict the covering relation associated with the order. Figure A.1 contains diagrams for a variety of ordered sets.

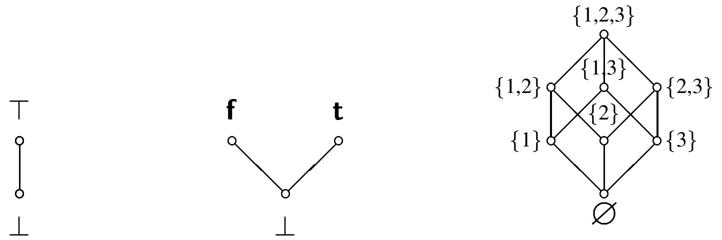


Figure A.1: Examples for partially ordered sets

Given partially ordered sets P and Q, a function $\varphi : P \to Q$ is *monotone* (or order-preserving) iff $x \sqsubseteq y$ implies $\varphi(x) \sqsubseteq \varphi(y)$, for all $x, y \in P$. The set of all monotone functions from P to Q is denoted by $P \to_m Q$. It is an *order-embedding* iff $x \sqsubseteq y$ if and only if $\varphi(x) \sqsubseteq \varphi(y)$, for all $x, y \in P$. The sets P and Q are *order-isomorphic* (notation: $P \cong Q$) iff there exist monotone maps $\varphi : P \to Q$ and $\psi : P \to Q$ such that $\varphi \circ \psi = \mathbf{id}$ and $\psi \circ \varphi = \mathbf{id}$.

Let P be an ordered set, $Q \subseteq P$ is a *down-set* (or *order ideal*) iff $x \in Q$ and $y \sqsubseteq x$ implies $y \in Q$, for all $x, y \in P$. Let $x \in P$, then $\downarrow x := \{\, y \in P \mid y \sqsubseteq x \,\}$ is a down-set. The set of all down-sets of P is denoted by $\mathcal{O}(P)$. The order duals of down-sets are *up-sets* (or *order filters*). The family of all up-sets is denoted by $\mathcal{F}(P)$. Note that $\mathcal{O}(P) \cong \mathcal{F}(P) \cong P \to_m \mathbf{2}$ where $\mathbf{2}$ is the two element order depicted in Figure A.1.

Let $Q \subseteq P$, an element $a \in Q$ is *minimal in* Q iff $x \sqsubseteq a$ implies $x = a$, for all $x \in Q$. The set of minimal elements of Q is denoted by $\operatorname{Min} Q$. The element $a \in Q$ is the *least element* of Q iff $a \sqsubseteq x$, for all $x \in Q$. *Maximal* and *greatest* elements are defined dually. The least element of a partial order P—if it exists—is called the *bottom element* and denoted by $\bot$ (pronounce: bottom). Accordingly, the greatest element is termed *top element* and written $\top$ (pronounce: top). A partially ordered set with a bottom element is called *pointed*.

Let $Q \subseteq P$, an element $a \in P$ is an *upper bound of* Q iff $x \sqsubseteq a$, for all $x \in P$. The set of all upper bounds is denoted by Q^u. If $a \in Q^u$ is the least element of Q^u, then a is called the *least upper bound* (or supremum). The least upper bound is clearly unique and is denoted by $\bigsqcup Q$. *Lower bounds* (notation: Q^ℓ) and *greatest lower bounds* (or infimum, notation: $\bigsqcap Q$) are defined dually.

A.1 Lemma *Let P be an ordered set, let $Q \subseteq P$ and let $a \in P$. Then a is the least upper bound of Q if and only if*

$$(\forall x \in P)\, ((\forall q \in Q)\ q \sqsubseteq x) \Longleftrightarrow a \sqsubseteq x.$$

Let P be an ordered set. Then $Q \subseteq P$ is a *chain* if either $x \sqsubseteq y$ or $y \sqsubseteq x$, for all $x, y \in Q$. The set $\{1, \ldots, n\}$ forms a chain $\mathbf{n}$ (the ordinal $\mathbf{n}$) in its natural order $\leqslant$. If $Q \subseteq P$ is a chain in P with $\operatorname{card} Q = n + 1$ we say that the length of Q is n. The order P is of length n (notation: $\operatorname{length} P = n$), if the length of the longest chain in P is n. A partial order satisfies the ascending chain condition if there is no infinite sequence $x_1 \sqsubset x_2 \sqsubset \cdots \sqsubset x_n \sqsubset \cdots$ of elements in P. The simplest order which violates this condition is the ordinal ω (the order type of the natural numbers in their natural order $\leqslant$).

A.2 Semilattices and lattices

Let P be an ordered set and let $x, y \in P$. The binary variants of $\bigsqcup$ and $\bigsqcap$ are defined by $x \sqcup y := \bigsqcup\{x, y\}$ (pronounce: *join*) and $x \sqcap y := \bigsqcap\{x, y\}$ (pronounce: *meet*). If $x \sqcup y$ exists in P for all $x, y \in P$, then P is called $\sqcup$-*semilattice*. If $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in P$, then P is called *lattice*. The ordered set $\langle \wp\{1, 2, 3\}; \subseteq\rangle$ depicted in Figure A.1 is a lattice while $\langle \mathbb{B}_\perp; \sqsubseteq\rangle$ is not.

The definition of $\sqcup$-semilattices and lattices are order-theoretical. Alternatively they may be characterized by algebraic properties.

A.2 Theorem *Let A be a non-empty set, let $+, * : A \times A \to A$ be binary operations on A and let $0 \in A$ be an element. Define $\preccurlyeq$ by $x \preccurlyeq y :\Longleftrightarrow x + y = y$. Then*

1. *$\preccurlyeq$ is reflexive iff $+$ is idempotent,*
2. *$\preccurlyeq$ is transitive iff $+$ is associative,*
3. *$\preccurlyeq$ is antisymmetric iff $+$ is commutative.*

Assume now that both $+$ and $*$ are idempotent, associative and commutative. Then

4. *$x + y$ is the join of x and y,*
5. *0 is the least element of A iff 0 is neutral with respect to $+$,*
6. *$x + y = y \Longleftrightarrow x = x * y$ iff $(x * y) + x = x$ and $x = x * (x + y)$ (absorption laws).*

Hence, $\langle A; \preccurlyeq\rangle$ is a $\sqcup$-semilattice if $+$ is idempotent, associative, commutative. If both operations are ACI and if they satisfy the absorption laws, then $\langle A; \preccurlyeq\rangle$ is a lattice.

Let L be a lattice with a least element 0. Then $a \in L$ is called an *atom* iff $0 \prec a$. The set of all atoms of L is denoted by $\mathcal{A}(L)$.

Given $\sqcup$-semilattices L and K, a function $\varphi : L \to K$ is called *additive* if $\varphi(x \sqcup y) = \varphi(x) \sqcup \varphi(y)$, for all $x, y \in L$.

A.3 Cpos, continuous functions and fixpoints

Let P be an ordered set. A non-empty subset S of P is said to be *directed* iff for every finite subset $F \subseteq S$, there exist $x \in S$ such that $x \in F^u$. An non-empty subset J of P is called an *ideal* iff it is a directed down-set. The family of all ideals of P is denoted by $\mathcal{I}(P)$.

A pointed ordered set D is a *complete partially ordered set* (cpo for short) iff $\bigsqcup S$ exists for every directed subset $S \subseteq D$. The bottom element $\perp$ of D is sometimes referred to as the improper element of D, while the elements of $D^\circ = D \setminus \{\perp\}$ are termed proper. The ordered sets depicted in Figure A.1 constitute cpos. A set S may be turned to a cpo (notation: $S_\perp$) by adjoining a least element and ordering the elements as follows: $x \sqsubseteq_{S_\perp} y \Longleftrightarrow x = \perp \vee x = y$. The cpo $S_\perp$ is termed the flat order associated with S.

Let D and E be cpos. A function $\varphi : D \to E$ is *continuous* iff $\varphi(\bigsqcup S) = \bigsqcup \varphi(S)$, for every directed set $S \subseteq D$. Continuity implies monotonicity, but the converse is not true in general. Given a function $\varphi : S_1 \times \cdots \times S_n \to T$ on sets there are generally several ways to extend φ to a continuous function $\varphi : (S_1)_\perp \times \cdots \times (S_n)_\perp \to T_\perp$. The logical disjunction, for example, considered as a function of type $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ with $\mathbb{B} = \{\mathbf{f}, \mathbf{t}\}$ gives rise to four different extensions.

or	$\bot$	**f**	**t**
$\bot$	$\bot$	$\bot$	$\bot$
f	$\bot$	**f**	**t**
t	$\bot$	**t**	**t**

sor	$\bot$	**f**	**t**
$\bot$	$\bot$	$\bot$	$\bot$
f	$\bot$	**f**	**t**
t	**t**	**t**	**t**

sor′	$\bot$	**f**	**t**
$\bot$	$\bot$	$\bot$	**t**
f	$\bot$	**f**	**t**
t	$\bot$	**t**	**t**

por	$\bot$	**f**	**t**
$\bot$	$\bot$	$\bot$	**t**
f	$\bot$	**f**	**t**
t	**t**	**t**	**t**

We will normally employ the least extension of a set-theoretic function ('or' in the example above). As another example consider the *conditional function* $(\cdot \longrightarrow \cdot \mid \cdot) : \mathbb{B}_\bot \times D \times D \to D$ defined by

$$
\begin{aligned}
(c \longrightarrow x \mid y) &= \bot \quad \textbf{if } c = \bot \\
&= x \quad \textbf{if } c = \mathbf{t} \\
&= y \quad \textbf{if } c = \mathbf{f}
\end{aligned}
$$

Since $(\mathbf{t} \longrightarrow \top \mid \bot) = \top$, the conditional $(\cdot \longrightarrow \cdot \mid \cdot)$ is not the least extension of its set-theoretic counterpart. Nevertheless this is in accordance with computational practice. We use $(c \longrightarrow x)$ as a shortcut for $(c \longrightarrow x \mid \bot)$.

Let D be an ordered set and let $\psi : D \to D$ be a map. An element $a \in D$ is termed *pre-fixpoint* iff $a \sqsubseteq \psi(a)$. Accordingly, a is a *fixpoint* iff $a = \psi(a)$. The following theorem due to Kleene guarantees the existence of least fixpoints. [Note that it slightly generalizes the original theorem since the least fixpoint greater than a given pre-fixpoint is determined.]

A.3 Fixpoint Theorem *Let D be a cpo, let $\psi : D \to D$ be continuous and let a be a pre-fixpoint of ψ. Then $\mathbf{lfp}_a(\psi)$ with*

$$\mathbf{lfp}_a(\psi) \quad = \quad \bigsqcup_{n \geqslant 0} \psi^n(a)$$

is the least fixpoint of ψ greater than a.

If ψ is strict then $\mathbf{lfp}_a(\psi)$ with $\bot \sqsubset a \sqsubseteq \psi(a)$ avoids the trivial fixpoint $\bot$. We use $\mathbf{lfp}$ as a shorthand for $\mathbf{lfp}_\bot$. Properties of the least fixpoint are established using *Scott's Principle of Fixpoint Induction*.

A.4 Fixpoint Induction Principle *Let D be a cpo, let $\psi : D \to D$ be continuous and let a be a pre-fixpoint of ψ. If $P \subseteq D$ satisfies*

1. *$a \in P$,*
2. *$x \in P \Longrightarrow \psi(x) \in P$, for all $x \in D$, and*
3. *$S \subseteq P \Longrightarrow \bigsqcup S \in P$, for all directed sets $S \subseteq D$,*

then $\mathbf{lfp}_a(\psi) \in P$.

Predicates which enjoy Property 3 in the list above are said to be *admissible* (meaning simply that they are admissible to Fixpoint Induction). Since this condition is often awkward to check, it is useful to know when it is satisfied automatically. Let $\varphi_i, \psi_i : D \to E$ for $1 \leqslant i \leqslant n$ be continuous functions. Then $P \subseteq D$ with $P(x) \iff (\forall i)\ \varphi_i(x) \sqsubseteq \psi_i(x)$ is admissible. If $P, Q \subseteq D$ are admissible so are $P \cap Q$ and $P \cup Q$. If $R \subseteq P \times Q$ is admissible with respect to its first argument then $(\forall y \in Q)\ R(x, y)$ is admissible as well.

A.4 Construction of cpos

A.4.1 Lifts

Let D be an partially ordered set, its lift $D_\perp$ is defined by

$$D_\perp \ := \ \{ (o, d) \mid d \in D \} \cup \{\perp\}$$

with the order

$$x \sqsubseteq_{D_\perp} y \iff x = \perp \vee (\exists x', y' \in D)\ x = (o, x') \wedge y = (o, y') \wedge x' \sqsubseteq_D y'.$$

Since every non-trivial directed set in $D_\perp$ essentially lies in D, the partially ordered set $D_\perp$ is a cpo if D is. Elements of $D_\perp$ are synthesized via $\textsf{\textbf{up}} : D \to D_\perp$ where $\textsf{\textbf{up}}(d) = (o, d)$ and analysed using $\textsf{\textbf{down}} : D_\perp \to D$ and $\partial : D_\perp \to \mathbb{B}_\perp$ ('termination predicate') given by

$$\begin{aligned} \textsf{\textbf{down}}(\perp) &= \perp & \qquad \partial(\perp) &= \perp, \\ \textsf{\textbf{down}}(o, d) &= d & \qquad \partial(o, d) &= \textsf{\textbf{t}}. \end{aligned}$$

[The termination predicate is defined accordingly for every complete partial order.] Note that **up**, **down** and ∂ are continuous. They satisfy

$$\textsf{\textbf{down}}(\textsf{\textbf{up}}\, x) = x,$$
$$x = (\partial x \longrightarrow \textsf{\textbf{up}}(\textsf{\textbf{down}}\, x)).$$

A cpo frequently used in the text is the (doubly) lifted domain of truth values $(\mathbb{B}_\perp)_\perp$.

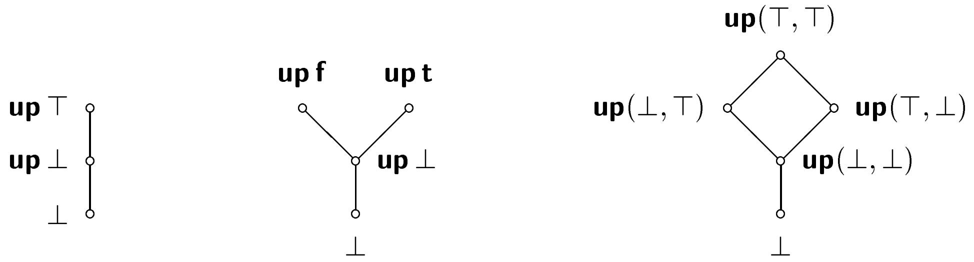


Figure A.2: The cpos $\langle \mathbf{3}; \sqsubseteq \rangle$, $\langle (\mathbb{B}_\perp)_\perp; \sqsubseteq \rangle$ and $\langle (\mathbf{2} \times \mathbf{2})_\perp; \sqsubseteq \rangle$

A.4.2 Sums

Let $D_1, \ldots D_n$ be partially ordered sets, then their *coalesced sum* (or amalgamated sum) $D_1 \oplus \cdots \oplus D_n$ is given by

$$D_1 \oplus \cdots \oplus D_n \ := \ \bigcup_{1 \leqslant i \leqslant n} \{ (i, d) \mid d \in D_i \wedge d_i \neq \perp \} \cup \{\perp\}$$

with the order

$$\begin{aligned} x \sqsubseteq_{D_1 \oplus \cdots \oplus D_n} y \iff & x = \perp \vee (\exists i)(\exists x', y' \in D_i) \\ & \qquad x = (i, x') \wedge y = (i, y') \wedge x' \sqsubseteq_{D_i} y'. \end{aligned}$$

Note that $D_1 \oplus \cdots \oplus D_n \cong \mathbf{1}$ for $n = 0$. The sum is a cpo if its constituents are, since all non-trivial directed sets essentially lie within one of the summands. Elements of $D_1 \oplus \cdots \oplus D_n$

are synthesized via the injection functions $\mathsf{in}_i : D_i \circ\!\!\to D_1 \oplus \cdots \oplus D_n$ where $\mathsf{in}_i(d) = (\partial d \longrightarrow (i, d))$ and analysed using $[\cdot, \ldots, \cdot] : (D_i \circ\!\!\to D) \times \cdots \times (D_n \circ\!\!\to D) \circ\!\!\to D_1 \oplus \cdots \oplus D_n \circ\!\!\to D$ ('case analysis') given by

$$\begin{aligned} [\varphi_1, \ldots, \varphi_n](\bot) &= \bot \\ [\varphi_1, \ldots, \varphi_n](i, d) &= \varphi_i(d). \end{aligned}$$

Note that $[\varphi_1, \ldots, \varphi_n] = \boldsymbol{\lambda} v.\bot$ for $n = 0$. Note, too, that both injection functions and case analysis are continuous. For better readability we will often use names rather than numerical indices for the constituents of a sum. Furthermore, we allow for an incomplete variant of case analysis: $[c_1 \colon \varphi_1, \ldots, c_n \colon \varphi_n]$ maps (c, d) to $\varphi_i(d)$ if $c_i = c$ and to $\bot$ otherwise. Note that we identify run-time errors such as 'missing case' with bottom. Let $\mathsf{is}_i = [i \colon \boldsymbol{\lambda} v.\mathbf{t}]$ and $\mathsf{out}_i = [i \colon \boldsymbol{\lambda} v.v]$, then

$$\mathsf{out}_i(\mathsf{in}_i\, x) = x$$
$$x = (\mathsf{is}_1(x) \longrightarrow \mathsf{in}_i(\mathsf{out}_i\, x) \mid \cdots \mid \mathsf{is}_n(x) \longrightarrow \mathsf{in}_n(\mathsf{out}_n\, x)).$$

We obtain a diagram for the ordered set $D_1 \oplus \cdots \oplus D_n$ by arranging side by side the diagrams for $D_1, \ldots, D_n$ with the bottom elements identified.

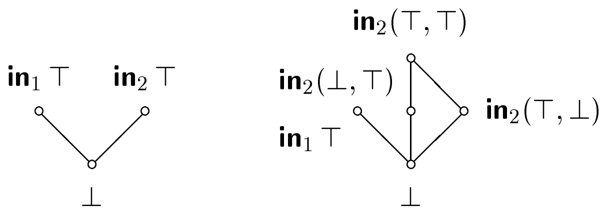


Figure A.3: The cpos $\langle \mathbf{2} \oplus \mathbf{2}; \sqsubseteq\rangle$ and $\langle \mathbf{2} \oplus \mathbf{2} \times \mathbf{2}; \sqsubseteq\rangle$

A.4.3 Products

Let $D_1, \ldots\ D_n$ be partially ordered sets, their *cartesian product*

$$D_1 \times \cdots \times D_n \quad := \quad \{\, (d_1, \ldots, d_n) \mid (\forall i)\ d_i \in D_i \,\}$$

may be turned to an ordered set by imposing the *coordinatewise order* defined by

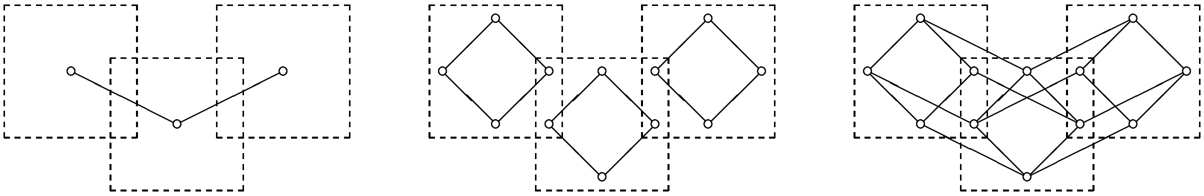
$$(x_1, \ldots, x_n) \sqsubseteq_{D_1 \times \cdots \times D_n} (y_1, \ldots, y_n) \iff (\forall i)\ x_i \sqsubseteq_{D_i} y_i.$$

Elements of $D_1 \times \cdots \times D_n$ are synthesized via $(\cdot, \ldots, \cdot)$ and analysed using the projection functions $\mathsf{on}_i : D_1 \times \cdots \times D_n \to D_i$ where $\mathsf{on}_i(x_1, \ldots, x_n) = x_i$. The tupling operator and the projection functions are continuous. They are related by

$$\mathsf{on}_i(d_1, \ldots, d_n) = d_i$$
$$d = (\mathsf{on}_1\, d, \ldots, \mathsf{on}_n\, d).$$

To see that $D_1 \times \cdots \times D_n$ is a cpo—if its constituents are—let $S \subseteq D_1 \times \cdots \times D_n$ be a directed set. Then $S_i = \mathsf{on}_i(S)$ is directed for $1 \leqslant i \leqslant n$ and $\bigsqcup S = (\bigsqcup S_1, \ldots, \bigsqcup S_n)$.

In order to draw a diagram for the cartesian product it is advisable to first 'reduce' $D_1 \times D_2 \times \cdots \times D_{n-1} \times D_n$ to the iterated binary product $D_1 \times (D_2 \times (\cdots (D_{n-1} \times D_n) \cdots))$. A diagram for $D \times E$ is obtained by replacing each point of a diagram for D by a diagram for E and connecting corresponding points. The picture below illustrates the main steps in drawing $\mathbb{B}_\bot \times (\mathbf{2} \times \mathbf{2})$.



In the text we will usually employ the *smash product* (or strict product) $D_1 \otimes \cdots \otimes D_n$ defined by

$$D_1 \otimes \cdots \otimes D_n \quad := \quad \{\, (d_1, \ldots, d_n) \mid (\forall i)\ d_i \in D_i \wedge d_i \neq \bot \,\} \cup \{\bot\}.$$

The ordering is coordinatewise stipulating that $\bot \sqsubseteq_{D_1 \otimes \cdots \otimes D_n} x$, for all $x \in D_1 \otimes \cdots \otimes D_n$. Note that $D_1 \otimes \cdots \otimes D_n \cong \mathbf{1}_\bot$ for $n = 0$. Elements of $D_1 \otimes \cdots \otimes D_n$ are constructed using the strict variant of tupling

$$\begin{array}{rcll} \langle d_1, \ldots, d_n \rangle & = & \bot & \textsf{\textbf{if}}\ (\exists i)\ d_i = \bot \\ & = & (d_1, \ldots, d_n) & \textsf{\textbf{otherwise}}. \end{array}$$

Strict tupling and the projection functions satisfy

$$\begin{array}{l} \textsf{\textbf{on}}_i\, \langle d_1, \ldots, d_n \rangle = (\partial d_1 \wedge \cdots \wedge \partial d_n \longrightarrow d_i) \\ d = \langle \textsf{\textbf{on}}_1\, d, \ldots, \textsf{\textbf{on}}_n\, d \rangle. \end{array}$$

Normal products and smash products are related by $D_\bot \otimes E_\bot \cong (D \times E)_\bot$. Hence, the smash product is drawn by removing the bottom elements, picturing the cartesian product and adding a bottom element.

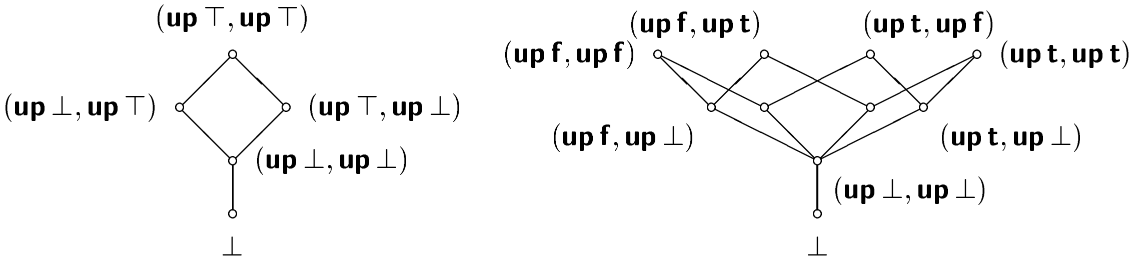


Figure A.4: The cpos $\langle \mathbf{2}_\bot \otimes \mathbf{2}_\bot ; \sqsubseteq \rangle$ and $\langle (\mathbb{B}_\bot)_\bot \otimes (\mathbb{B}_\bot)_\bot ; \sqsubseteq \rangle$

A.4.4 Exponents

Let D and E be ordered sets, the set $D \to E$ of all functions from D to E may be turned to an ordered set by imposing the *pointwise order* defined by

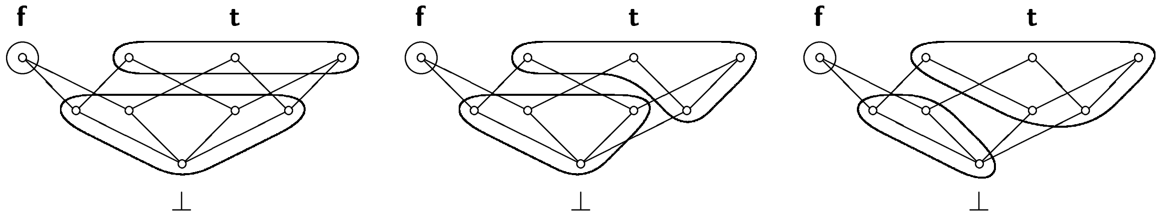
$$\varphi \sqsubseteq_{D \to E} \psi \quad \Longleftrightarrow \quad (\forall x \in D)\ \varphi(x) \sqsubseteq_E \psi(x).$$

Let D and E be cpos, then the set of all continuous functions from D to E with the pointwise order is denoted by $[D \to E]$. Note that we will often be sloppy and omit the square brackets. If $\Phi \subseteq [D \to E]$ is directed, then $\Psi(x) := \{\, \varphi(x) \mid \varphi \in \Phi \,\}$ is directed, for all $x \in D$. Hence the pointwise join $(\bigsqcup \Phi)(x) = \bigsqcup \Phi(x)$ is well-defined and since $\bigsqcup \Phi$ is continuous, $[D \to E]$ is a cpo.

We will use occasionally typed λ-expressions to denote elements of $[D \to E]$. Let v be a variable ranging over D and α be a formal expression of type E, then $\boldsymbol{\lambda} v.\alpha$ (λ-abstraction) denotes the continuous function $\varphi \in [D \to E]$ with $\varphi(v) = \alpha$.

Let $\varphi \in [D \to E]$, then φ is called strict iff $\varphi(\bot) = \bot$, that is, φ preserves bottom. The set of all strict continuous functions from D to E is denoted by $[D \circ\!\!\to E]$. The strict variant of λ-abstraction $\boldsymbol{\lambda}^{\circ} v.\alpha = \boldsymbol{\lambda} v.(\partial v \longrightarrow \alpha)$ denotes an element of $[D \circ\!\!\to E]$. Using the strict λ-abstraction we may define $\textsf{\textbf{strict}} = \boldsymbol{\lambda}\varphi.\boldsymbol{\lambda}^{\circ} v.\varphi(v)$. Continuous functions and strict continuous functions are related by $[D \to E] \cong [D_\bot \circ\!\!\to E]$. The isomorphism is established by the maps $(\cdot)^\dagger : [D \to E] \to [D_\bot \circ\!\!\to E]$ and $(\cdot)^\ddagger : [D_\bot \circ\!\!\to E] \to [D \to E]$ where $\varphi^\dagger = \textsf{\textbf{strict}}(\varphi \circ \textsf{\textbf{down}})$ and $\varphi^\ddagger = \varphi \circ \textsf{\textbf{up}}$ respectively.

Though the need of drawing $[D \to E]$ does not arise it will often prove helpful to depict elements of $[D \to E]$. Let $\varphi \in [D \to E]$, a pictorial representation of φ is obtained by drawing a diagram for D and labelling the points with their image under φ. For better readability we will normally encircle points having a common image. Take as an example the continuous extensions of 'logical or' tabulated in Section A.3.



or sor por

A.5 Complete semilattices and lattices

Let P be a non-empty set. A non-empty subset S of P is *bounded* (or consistent) iff $S^u \neq \varnothing$. If $\bigsqcup S$ exists for every bounded subset $S \subseteq P$, then P is called a *complete semilattice* (or bounded complete cpo). If P is a complete semilattice, then $\bigsqcap S$ exists for $\varnothing \neq S \subseteq P$.

Adjoining a top element to a complete semilattice creates a *complete lattice*, that is, a partial order where $\bigsqcup S$ and $\bigsqcap S$ exist for arbitrary sets.

Many examples of complete lattices arise as topped $\bigcap$-structures on some set. Let X be a set and let $\mathfrak{L}$ be a non-empty family of subsets of X ordered by set inclusion. If $\bigcap_{i \in I} A_i \in \mathfrak{L}$ for every non-empty family $\{A_i\}_{i \in I} \subseteq \mathfrak{L}$, then $\mathfrak{L}$ is called an $\bigcap$-*structure* (or Moore family) on X. If additionally $X \in \mathfrak{L}$, we say that $\mathfrak{L}$ is a *topped* $\bigcap$-*structure* on X. The set of down-sets $\mathcal{O}(P)$ of P, for example, forms a topped $\bigcap$-structure on P.

A.5 Theorem *Let $\mathfrak{L}$ be an $\bigcap$-structure on X. Then $\mathfrak{L}$ is a complete semilattice in which*

$$\begin{aligned} \bigsqcap_{i \in I} A_i &= \bigcap_{i \in I} A_i, \\ \bigsqcup_{i \in I} A_i &= \bigcap \{\, B \in \mathfrak{L} \mid \bigcup_{i \in I} A_i \subseteq B \,\}. \end{aligned}$$

If $\mathfrak{L}$ is topped, then $\mathfrak{L}$ forms a complete lattice.

Each topped $\bigcap$-structure on X induces a closure operator on $\wp(X)$. A map $C : \wp(X) \to \wp(X)$ is a *closure operator* iff it is monotone, idempotent ($C(C(A)) = C(A)$) and is extensive ($C(A) \supseteq A$).

A.6 Theorem *Let C be a closure operator on a set X. The family*

$$\mathfrak{L}_C \;:=\; \{\, A \subseteq X \mid C(A) = A \,\}$$

is a topped $\bigcap$-structure on X in which

$$\begin{aligned} \bigsqcap_{i\in I} A_i &= \bigcap_{i\in I} A_i, \\ \bigsqcup_{i\in I} A_i &= C(\bigcup_{i\in I} A_i). \end{aligned}$$

Conversely, given a topped $\bigcap$-structure $\mathfrak{L}$ on X

$$C_{\mathfrak{L}}(A) \;:=\; \bigcap\{\, B \in \mathfrak{L} \mid A \subseteq B \,\}$$

is a closure operator on X.

The closure operator associated with $\mathcal{O}(P)$ is $\downarrow\cdot$ given by $\downarrow X = \{\, y \mid x \in X \wedge y \sqsubseteq x \,\}$.

The concept of a closure operator may be generalized to arbitrary complete lattices. Let L be a complete lattice and let $c : L \to L$ be a closure operator on L, then the set $L_c := \{\, a \in L \mid c(a) = a \,\}$ is a complete lattice in which $\bigsqcap_{L_c} X = \bigsqcap_L X$ and $\bigsqcup_{L_c} X = c(\bigsqcup_L X)$, for all $X \subseteq L_c$.

A.6 Algebraic semilattices and lattices

Let D be a cpo and let $d \in D$. Then d is called *finite* (or isolated or compact) in D iff $d \sqsubseteq \bigsqcup S$ implies $(\exists s \in S)\ d \sqsubseteq s$, for every directed set $S \subseteq D$. The set of finite elements of D is denoted $F(D)$. Furthermore, $F^\circ(D) = F(D) \setminus \{\bot\}$. The set $F(D)$ is closed under finite joins—provided they exist. The finite elements of $\wp(X)$, for example, are the finite subsets $F \subseteq X$ of X (notation: $F \Subset X$). On the other hand the set of real numbers, $\mathbb{R}$, contains no finite elements since $r = \bigsqcup\{\, s \in \mathbb{R} \mid s < r \,\}$ for all $r \in \mathbb{R}$.

A cpo D is said to be *algebraic* iff, for every $x \in D$, the set $F(D) \cap \downarrow x$ is directed and $x = \bigsqcup F(D) \cap \downarrow x$. The set of finite elements $F(D)$ is termed the *basis* of D. If D is a complete semilattice and algebraic, then we will call D a *domain*. In this case the set $F(D) \cap \downarrow x$ is trivially directed. [In connection with computability questions the set of finite elements is restricted to be countable. The domains used in this work satisfy this condition though no essential use is made of this fact.] Flat cpos provide simple examples for domains. The constructions $(\cdot)_\bot$, $\oplus$, $\times$, $\otimes$, $\to$, and, $\circ\!\!\to$ preserve bounded completeness and algebraicity. We have, for example, $F(D_\bot) = F(D)_\bot$, $F(D_1 \oplus \cdots \oplus D_n) = F(D_1) \oplus \cdots \oplus F(D_n)$, and $F(D_1 \otimes \cdots \otimes D_n) = F(D_1) \otimes \cdots \otimes F(D_n)$. Since every domain equation involving these constructions yields a domain, this class of spaces is well-suited for the purposes of semantic definitions.

A continuous function on an algebraic cpo is determined by its effect on the finite elements.

A.7 Theorem *Let D and E be algebraic cpos and $\varphi : D \to E$ be monotone. Then the following are equivalent:*

1. *φ is continuous,*

2. *$(\forall x \in D)\ \varphi(x) = \bigsqcup \varphi(F(D) \cap \downarrow x)$,*

3. $(\forall x \in D)(\forall e \in F(E) \cap \downarrow\varphi(x))(\exists d \in F(D) \cap \downarrow x)\ e \sqsubseteq \varphi(d)$. *($\epsilon$-$\delta$-definition).*

Property 3 may be rephrased as follows: To obtain a finite amount of information about $\varphi(x)$ it is only necessary to input a finite amount of information about x. Using the ϵ-δ-definition of continuity it is, for example, immediate that the binary join is continuous—if it always exists as in complete semilattices. Theorem A.7 furthermore implies that $[D \to E]$ is isomorphic to $F(D) \to_m E$.

The correspondence between complete semilattices and $\bigcap$-structure specializes to the algebraic case. An $\bigcap$-structure is called algebraic iff $\bigcup_{i\in I} A_i \in \mathfrak{L}$ for every directed family $\{A_i\}_{i\in I} \subseteq \mathfrak{L}$. A closure operator is algebraic iff $C(\bigcup_{i\in I} A_i) = \bigcup_{i\in I} A_i$ for every directed family $\{A_i\}_{i\in I} \subseteq \mathfrak{L}$. [Note that C is algebraic iff C is continuous.] As to be expected, a closure operator C is algebraic iff $\mathfrak{L}_C$ is an algebraic $\bigcap$-structure and vice versa. The set of all down-sets $\mathcal{O}(P)$, for example, is algebraic with $F(\mathcal{O}(P)) = \{\downarrow F \mid F \Subset P\}$.

A.8 Theorem *Let $\mathfrak{L}$ be a (topped) algebraic $\bigcap$-structure. Then $\mathfrak{L}$ is a domain (algebraic lattice).*

An ordered set can be embedded into a cpo using the *ideal completion*.

A.9 Theorem *Let P be a partial order with a least element. Then $\langle \mathcal{I}(P); \subseteq\rangle$ is an algebraic cpo with $F(\mathcal{I}(P)) = \{\downarrow x \mid x \in P\}$. The map $x \mapsto \downarrow x$ is an order-embedding of P into $\mathcal{I}(P)$. If $\bigsqcup F$ exists in P for all bounded (arbitrary) finite sets $F \Subset P$, then $\mathcal{I}(P)$ is a (topped) algebraic $\bigcap$-structure.*

Each domain may be realized as an algebraic $\bigcap$-structure.

A.10 Theorem *Let D be a domain (algebraic lattice). Then $\mathcal{I}(F(D))$ is a (topped) algebraic $\bigcap$-structure isomorphic to D. The map $x \mapsto F(D) \cap \downarrow x$ is an isomorphism of D onto $\mathcal{I}(F(D))$ with its inverse given by $J \mapsto \bigsqcup J$.*

Let D be a domain, then $\varnothing \neq S \subseteq D$ is a *Scott-closed set* (or ideal[1]) iff S is a down-set and closed under directed joins, that is, $X \subseteq S$ implies $\bigsqcup X \in S$, for all directed sets $X \subseteq D$. A set is a *Scott-open set* iff it is the complement of a Scott-closed set. The set of all Scott-closed subsets of D is denoted $\mathcal{P}(D)$ and amounts to the Hoare (or lower or partial correctness) powerdomain.

A.11 Theorem *Let D be a domain. Then $X \mapsto X \cap F^\circ(D)$ is an order-isomorphism of $\mathcal{P}(D)$ onto $\mathcal{O}(F^\circ(D))$, with its inverse given by $X \mapsto \{\bigsqcup Y \mid Y \subseteq X$ and Y directed$\} \cup \{\bot\}$.*

The theorem immediately implies that $\langle \mathcal{P}(D); \subseteq\rangle$ is an algebraic lattice. Furthermore, $\mathcal{P}(D)$ is isomorphic to $[D^\circ \to \mathbf{2}]$ since $\mathcal{P}(D) \cong \mathcal{O}(F^\circ(D)) \cong F^\circ(D) \to_m \mathbf{2} \cong [D^\circ \to \mathbf{2}]$.

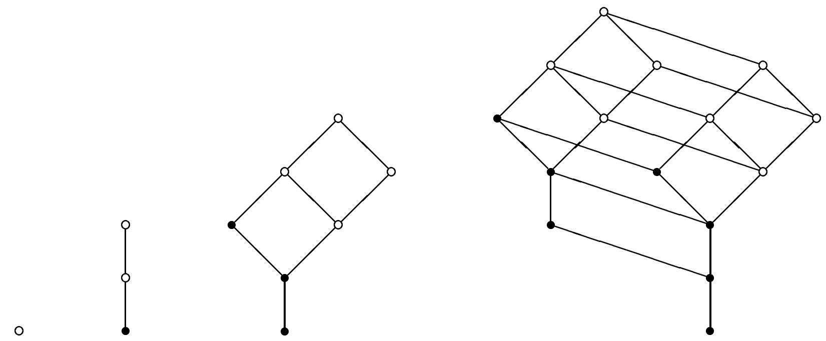
A.7 Projections and solving recursive domain equations

The purpose of this section is twofold. We show how to construct solutions to recursive domain equations which appear, for example, in the standard semantics presented in Chapter 3. [The results pave the way for the next section where we prove that solutions to first-order equations satisfy Axiom I.] Of the various methods which have been proposed we employ least fixpoints of *projections* on universal domains, see [64]. This particular choice allows

[1] The term ideal is a bit overloaded since each order structure (ordered set, cpo, lattice, etc) has its own concept of ideal.

us to review the basic mathematical facts about projections which are employed anyway in this dissertation. Since the material presented is used quite intensively we decided to provide proofs for most of the results.

The construction of solutions to recursive domain equations is, of course, guided by the Fixpoint Theorem. Consider the domain equation $D \cong \Phi(D)$ with $\Phi(D) = \mathbf{2}_\perp \otimes D_\perp$. A solution of this equation may be constructed as the limit of the chain $\mathbf{1}$, $\Phi(\mathbf{1})$, $\Phi^2(\mathbf{1}) \ldots$ of approximations. The first four elements of the chain are displayed below.



D_0 D_1 D_2 D_3

The shaded elements indicate that each domain D_{i+1} contains D_i as a subdomain. Insofar D_i actually approximates D_{i+1}. To make this idea mathematically precise we represent domains by finitary projections.

A.12 Definition *Let D be a domain. A continuous function $\pi : D \to D$ is called a* projection *iff it is idempotent ($\pi \circ \pi = \pi$) and reductive ($\pi \sqsubseteq \boldsymbol{\lambda} v.v$). A projection π is termed* finitary *iff $\mathbf{im}\,\pi := \{\, \pi\, v \mid v \in D \,\}$ is a domain. The set of all finitary projections on D is denoted $\|D\|$.*

It is not easy to construct a projection which is not finitary. Let $D = \mathcal{I}(\mathbb{Q}^+)$ where $\mathbb{Q}^+ := \{\, q \in \mathbb{Q} \mid q \geqslant 0 \,\}$. Theorem A.9 implies that D forms an algebraic lattice. Then $\pi : D \to D$ with

$$\pi(S) \quad = \quad \{\, q \in \mathbb{Q}^+ \mid q < \textstyle\bigsqcup_{\mathbb{R}} S \,\}$$

defines a projection on D. However, π is not finitary, since $\mathbf{im}\,\pi \cong \mathbb{R}^+$, that is, $\mathbf{im}\,\pi$ contains no finite elements at all.

We will show in the sequel that the set of finitary projections on a domain forms an algebraic lattice. This result allows us to invoke the 'ordinary' Fixpoint Theorem for solving domain equations. The following lemma will be handy.

A.13 Lemma *Let π be a projection on D, let $S \subseteq \mathbf{im}\,\pi$ and let $v \in \mathbf{im}\,\pi$.*

1. *If $\bigsqcup S$ exists in D, we have $\bigsqcup S \in \mathbf{im}\,\pi$. If D is a complete semilattice, so is $\mathbf{im}\,\pi$.*

2. *The element v is finite in $\mathbf{im}\,\pi$ iff v is finite in D, that is, $F(\mathbf{im}\,\pi) = F(D) \cap \mathbf{im}\,\pi$.*

Proof Ad 1) Since π is reductive we have $\pi(\bigsqcup S) \sqsubseteq \bigsqcup S$. Monotonicity implies $\pi(\bigsqcup S) \sqsupseteq \bigsqcup \pi(S) = \bigsqcup S$. If S is bounded in $\mathbf{im}\,\pi$, $\bigsqcup S$ exists in D and by the first part in $\mathbf{im}\,\pi$ as well.

Ad 2) '$\Longrightarrow$': Assume that $v \in F(\mathbf{im}\,\pi)$ and that $v \sqsubseteq \bigsqcup S$, where S is directed in D. Then $\pi(S)$ is directed in $\mathbf{im}\,\pi$ and there is an element $\pi(s) \in \pi(S)$ such that $v \sqsubseteq \pi(s)$. Hence $v \sqsubseteq \pi(s) \sqsubseteq s$, so $v \in F(D)$. '$\Longleftarrow$': Follows immediately since $\mathbf{im}\,\pi$ is a subset of D. ∎

The proof that $\|D\|$ is an algebraic lattice is based on the characterization of finitary projections in terms of so called normal subsets of $F(D)$. The following definitions and propositions introduce the necessary prerequisites. We first observe that a finitary projection is uniquely determined by the finite elements of its image. Assume that $N \subseteq F(D)$, then π_N with

$$\pi_N(v) \quad = \quad \bigsqcup N \cap {\downarrow} v$$

is the least finitary projection such that $N \subseteq F(\mathbf{im}\,\pi)$. Note that π_N is well-defined, since $N \cap {\downarrow} v$ is bounded.

A.14 Lemma *Let D be a domain and let $N \subseteq F(D)$. Then π_N is a finitary projection with $N \subseteq F(\mathbf{im}\,\pi_N)$.*

Proof A routine check confirms that π_N is reductive, idempotent and monotone. Since D is algebraic, a monotone function $\varphi : D \to D$ is continuous iff $\varphi(v) = \bigsqcup \varphi(F(D) \cap {\downarrow} v)$, for every $v \in D$. Note that for $v \in N$ we have $\pi_N(v) = \bigsqcup N \cap {\downarrow} v = v$, hence $N \subseteq F(\mathbf{im}\,\pi)$.

$$\pi_N(v) = \bigsqcup N \cap {\downarrow} v = \bigsqcup \pi_N(N \cap {\downarrow} v) \sqsubseteq \bigsqcup \pi_N(F(D) \cap {\downarrow} v)$$

Since D is algebraic, we have $v = \bigsqcup F(D) \cap {\downarrow} v$, so the reverse inclusion follows by monotonicity of π_N. It remains to show that $\mathbf{im}\,\pi_N$ is a domain. By Lemma A.13 $\mathbf{im}\,\pi_N$ is a complete semilattice. Furthermore, $\mathbf{im}\,\pi_N$ is algebraic, since for each $v \in \mathbf{im}\,\pi_N$

$$v = \pi_N(v) = \bigsqcup N \cap {\downarrow} v \sqsubseteq \bigsqcup F(\mathbf{im}\,\pi_N) \cap {\downarrow} v \sqsubseteq v. \qquad \blacksquare$$

However, not every subset of $F(D)$ corresponds to a finitary projection (see Section 4.1 for an example). A one-to-one correspondence is established if we confine ourselves to normal subsets.

A.15 Definition *Let P be an ordered set and let $N \subseteq P$. Then N is said to be* normal *in P (notation: $N \lhd P$) iff the set $N \cap {\downarrow} v$ is directed for every $v \in P$. The set of all normal substructures of P is denoted by $\mathfrak{N}(P)$.*

A.16 Theorem *Let D be a domain. Then the map $\pi \mapsto F(\mathbf{im}\,\pi)$ is an order-isomorphism of $\|D\|$ onto $\mathfrak{N}(F(D))$, with its inverse given by $N \mapsto \pi_N$.*

Proof Let $N \lhd F(D)$. Lemma A.14 implies that π_N is a finitary projection. It remains to show that $F(\mathbf{im}\,\pi_N) = N$. Assume that $v \in \mathbf{im}\,\pi_N$ is finite in D. Hence $v = \pi_N(v) = \bigsqcup N \cap {\downarrow} v$. Invoking the finiteness condition on the directed set $N \cap {\downarrow} v$ yields an element $w \in N \cap {\downarrow} v$ such that $v \sqsubseteq w$. Thus $v = w \in N$ and consequently $\mathbf{im}\,\pi_N \cap F(D) \subseteq N$. Since $N \subseteq \mathbf{im}\,\pi_N$, we have $F(\mathbf{im}\,\pi_N) = N$ by Lemma A.13.

Conversely, let $\pi \in \|D\|$. We show that $F(\mathbf{im}\,\pi)$ is normal and that $\pi_{F(\mathbf{im}\,\pi)} = \pi$. Let $v \in F(D)$ and $F \Subset F(\mathbf{im}\,\pi) \cap {\downarrow} v$. As F is bounded, $\bigsqcup F$ exists and since $F(D)$ is closed under finite joins—provided they exist—we have $\bigsqcup F \in F(D)$. Using Lemma A.13 we have $\bigsqcup F \in F(\mathbf{im}\,\pi) \cap {\downarrow} v$. Consequently, $F(\mathbf{im}\,\pi) \cap {\downarrow} v$ is directed and $F(\mathbf{im}\,\pi)$ is normal in $F(D)$. Because π is finitary we have

$$\pi(v) = \bigsqcup F(\mathbf{im}\,\pi) \cap {\downarrow} \pi(v) \sqsubseteq \bigsqcup F(\mathbf{im}\,\pi) \cap {\downarrow} v = \pi_{F(\mathbf{im}\,\pi)}(v).$$

On the other hand, continuity and idempotence implies

$$\pi(v) = \bigsqcup \pi(F(D) \cap {\downarrow} v) \sqsupseteq \bigsqcup \pi(F(\mathbf{im}\,\pi) \cap {\downarrow} v) = \bigsqcup F(\mathbf{im}\,\pi) \cap {\downarrow} v.$$

Finally, the maps $\pi \mapsto F(\mathbf{im}\,\pi)$ and $N \mapsto \pi_N$ are monotone and mutually inverse bijections. $\blacksquare$

The following lemma is essentially a restatement of Lemma A.13 in terms of normal substructures.

A.17 Lemma *Let $N \lhd F(D)$ and let $F \Subset N$. If $\bigsqcup F$ exists in D, we have $\bigsqcup F \in N$.*

Proof Let $v = \bigsqcup F$. Since $N \cap \downarrow v$ is directed, there exists an $w \in N \cap \downarrow v$ with $w \in F^u$. Hence $\bigsqcup F = w \in N$. ■

Note that the restriction to finite subsets of N is necessary to ensure that the join is finite, as well. Using the correspondence between finitary projections on D and normal subsets of $F(D)$ we have the following theorem.

A.18 Theorem *Let D be a domain. The set $\|D\|$ of finitary projections on D forms an algebraic lattice.*

Proof Using Theorem A.16 it suffices to show that $\langle \mathfrak{N}(F(D)); \subseteq \rangle$ is an algebraic lattice. We prove in fact that $\mathfrak{N}(F(D))$ is a topped algebraic $\bigcap$-structure.

First note, that $\downarrow v$ is directed, for all $v \in F(D)$; hence $F(D) \in \mathfrak{N}(F(D))$.

Let $\{N_i\}_{i \in I}$ be a non-empty family in $\mathfrak{N}(F(D))$. Take $v \in F(D)$ and $F \Subset (\bigcap_{i \in I} N_i) \cap \downarrow v$. Since F is bounded, $\bigsqcup F$ exists and by Lemma A.17 we have $\bigsqcup F \in N_i$, for all $i \in I$. Hence we may conclude that $\bigsqcup F \in (\bigcap_{i \in I} N_i) \cap \downarrow v$. Thus $\mathfrak{N}(F(D))$ is closed under intersection.

Let $\{N_i\}_{i \in I}$ be a directed family in $\mathfrak{N}(F(D))$. Take $v \in D$ and $F \Subset (\bigcup_{i \in I} N_i) \cap \downarrow v$. As $\{N_i\}_{i \in I}$ is directed, there exists an k such that $F \subseteq N_k$. Lemma A.17 yields $\bigsqcup F \in N_k$ and consequently $\bigsqcup F \in (\bigcup_{i \in I} N_i) \cap \downarrow v$. Thus the union of a directed family of normal subsets is normal as well. ■

Theorem A.18 enables us the invoke the Fixpoint Theorem for solving domain equations provided we succeed in recasting the domain operators in terms of projections.

A.19 Definition *An operator Φ on domains is* representable *over a domain U iff there is a function $R_\Phi \in \|U\| \times \cdots \times \|U\| \to \|U\|$ such that*

$$\mathbf{im}(R_\Phi(\pi_1, \ldots, \pi_n)) \quad \cong \quad \Phi(\mathbf{im}\,\pi_1, \ldots, \mathbf{im}\,\pi_n),$$

for each $\pi_1, \ldots \pi_n \in \|U\|$.

A.20 Theorem *If Φ is representable over a domain U, then there is a domain D such that $D \cong \Phi(D)$.*

Proof Suppose R_Φ represents Φ. The Fixpoint Theorem certifies the existence of $\pi \in \|U\|$ satisfying $\pi = R_\Phi(\pi)$. Since R_Φ represents Φ, we have $\mathbf{im}\,\pi = \mathbf{im}(R_\Phi(\pi)) \cong \Phi(\mathbf{im}\,\pi)$. Hence $\mathbf{im}\,\pi$ is the desired solution. ■

The main mathematical difficulty is to find a suitable domain over which the domain operators may be represented. Since its construction is rather involved we state without proof the following characterization of the universal domain **U** [64].

A.21 Theorem *For any domain D there is a projection-embedding pair $\phi : \mathbf{U} \to D$ and $\psi : D \to \mathbf{U}$, such that $\phi \circ \psi = \boldsymbol{\lambda} v.v$ and $\psi \circ \phi \sqsubseteq \boldsymbol{\lambda} v.v$.*

Using the theorem above it is not hard to show that the 'standard' domain operators are in fact representable on **U**.

A.22 Lemma *Let Φ be an operator built up from constant domains and using the domain constructors $\circ\!\!\rightarrow$, $\otimes$, $\oplus$, and $(\cdot)_\perp$. Then Φ is representable over $\mathbf{U}$.*

Proof Since the composition of representable operators is representable, the proposition follows by an induction over the 'term structure' of Φ. We confine ourselves to show the proposition for the domain operator $\oplus$. By Theorem A.20 we know that there is a projection-embedding pair $\phi_\oplus : \mathbf{U} \to \mathbf{U} \oplus \mathbf{U}$ and $\psi_\oplus : \mathbf{U} \oplus \mathbf{U} \to \mathbf{U}$. We claim that $R_\oplus : \|\mathbf{U}\| \times \|\mathbf{U}\| \to \|\mathbf{U}\|$ with

$$R_\oplus(\pi_1, \pi_2) \quad = \quad \psi_\oplus \circ (\pi_1 \oplus \pi_2) \circ \phi_\oplus$$

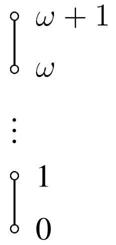
represents $\oplus$ on $\mathbf{U}$. The isomorphism between $\mathsf{im}(R_\oplus(\pi_1, \pi_2))$ and $\mathsf{im}\,\pi_1 \oplus \mathsf{im}\,\pi_2$ is established by the maps $\phi_\oplus$ and $\psi_\oplus$. Since $\phi_\oplus \circ \psi_\oplus = \boldsymbol{\lambda} v.v$, it suffices to show $(\psi_\oplus \circ \phi_\oplus)(v) = v$, for $v \in \mathsf{im}(R_\oplus(\pi_1, \pi_2))$.

$$\begin{aligned} (\psi_\oplus \circ \phi_\oplus)(v) &= (\psi_\oplus \circ \phi_\oplus)(R_\oplus(\pi_1, \pi_2)(v)) \\ &= (\psi_\oplus \circ \phi_\oplus \circ \psi_\oplus \circ (\pi_1 \oplus \pi_2) \circ \phi_\oplus)(v) \\ &= (\psi_\oplus \circ (\pi_1 \oplus \pi_2) \circ \phi_\oplus)(v) \\ &= v \end{aligned}$$

■

A.8 I-domains and stable functions

There are several intuitive but wrong ideas about the finite elements of a domain. One idea states that the height of a finite element is finite as well. The domain $\omega + 1$ provides a simple counterexample.



The element ω is a limit element while $\omega + 1$ is finite. The example furthermore shows that approximations of a finite element must not necessarily be finite as well. Both 'problems' disappear if we further restrict the class of domains to so called I-domains (domains satisfying Axiom I).

A.23 Definition *Let D be a domain, D is an I-domain iff*

$$(\forall d \in F(D)) \downarrow d \Subset D \qquad \textit{(Axiom } I\textit{).}$$

A finite element of an I-domain has only finitely many approximations, which implies that its height is finite. Since an element $v \in D$ with $F(D) \cap \downarrow v \Subset D$ is finite, Axiom I is equivalent to both $(\forall d \in F(D))\, F(D) \cap \downarrow d \Subset D$ and $(\forall d \in F(D)) \downarrow d \Subset F(D)$. Hence each approximation of a finite element is finite as well. Counterexamples typically involve the function space: The constant function $\varphi(n) = \top$ is finite but $\downarrow \varphi$ is infinite.

Theorem A.24 below shows that every domain which is obtained as the solution of a *first-order* domain equation does indeed satisfy this condition. It is essentially based on the observation that the embedding of $\Phi^i(\mathbf{1})$ into $\Phi^{i+1}(\mathbf{1})$ is downward closed (cf to the picture on page 175).

A.24 Theorem *Let Φ be an operator built up from constant I-domains and using the domain constructors $\otimes$, $\oplus$, and $(\cdot)_\perp$. Then D with $D \cong \Phi(D)$ is an I-domain.*

Proof Suppose R_Φ represents Φ. Then $D = \mathbf{im}(\bigsqcup_{i\geqslant 0} \pi_i)$ with $\pi_i = R^i_\Phi(\perp\!\!\!\perp)$ is a fixpoint of Φ.

We first show that each domain $D_i = \mathbf{im}\,\pi_i$ satisfies Axiom I. Consider the domain $D\otimes E$ and assume that $\downarrow d \Subset D$ and $\downarrow e \Subset E$, for finite elements $d \in F(D)$ and $e \in F(E)$. Now, let $v \in F(D\otimes E) = F(D)\otimes F(E)$, then $\downarrow v = \downarrow \mathbf{on}_1(v) \times \downarrow \mathbf{on}_2(v)$ is finite as well. Analogous results hold for $\oplus$ and $(\cdot)_\perp$. Hence each operator preserves Axiom I.

We show next that $F(D) \cap \downarrow a$ with $a \in F(D)$ is contained in $F(D_k)$, for some $k \in \mathbb{N}$. The characteristic property of operators constructed from $\otimes$, $\oplus$, $(\cdot)_\perp$, and constant domains is the following. Let $E = \mathbf{im}\,\pi$ and $F = \mathbf{im}(R_\Phi(\pi))$, then for each $v, w \in F$,

$$\begin{aligned}\perp \sqsubset v \sqsubseteq w \iff & (\exists \varphi : E\times\cdots\times E \to F)(\exists v_1,\ldots,v_n \in E)(\exists w_1,\ldots,w_n \in E)\\ & v = \varphi(v_1,\ldots,v_n) \wedge w = \varphi(w_1,\ldots,w_n) \wedge (\forall i)\, v_i \sqsubseteq w_i\end{aligned}$$

The functions φ whose existence is certified correspond roughly to the constructor functions of a data type definition. [Order-theoretically they are just embeddings.] This property allows us to prove that the inclusion of D_i in D_{i+1} is closed under going down: Let $a \in D_i \subseteq D_{i+1}$, then $D_{i+1} \cap \downarrow a \subseteq D_i$, for all $i \in \mathbb{N}$. Induction basis: Holds trivially. Induction step: Let $\perp \sqsubset v \in D_{i+2} \cap \downarrow a$, then there is a function φ, elements $v_1,\ldots,v_n \in D_{i+1}$ and $a_1,\ldots,a_n \in D_i \subseteq D_{i+1}$ with $v = \varphi(v_1,\ldots,v_n)$, $a = \varphi(a_1,\ldots,a_n)$ and $v_i \sqsubseteq a_i$. Applying the induction hypothesis to each a_i, we have $v_i \in D_i$ and hence $v = \varphi(v_1,\ldots,v_n) \in D_{i+1}$. Using another mathematical induction we get: Let $a \in D_k \subseteq D_i$, then $D_i \cap \downarrow a \subseteq D_k$, for all $i \geqslant k$. Since the chain $\{\pi_i\}_{i\geqslant 0}$ is directed we have by Theorem A.18

$$F(D) = F(\mathbf{im}(\bigsqcup_{i\geqslant 0} \pi_i)) = \bigcup_{i\geqslant 0} F(\mathbf{im}\,\pi_i) = \bigcup_{i\geqslant 0} F(D_i). \tag{A.1}$$

Now, let $a \in F(D)$. By (A.1) there is a $k \in \mathbb{N}$ with $a \in F(D_k)$. Furthermore,

$$\begin{aligned}(\bigcup_{i\geqslant 0} F(D_i)) \cap \downarrow a &= \bigcup_{i\geqslant 0} F(D_i) \cap \downarrow a && \cap \text{ distributes over } \bigcup\\ &= \bigcup_{i\geqslant 0} F(\mathbf{U}) \cap D_i \cap \downarrow a && \text{Lemma A.13}\\ &\subseteq F(\mathbf{U}) \cap D_k && \text{see above}\\ &= F(D_k) && \text{Lemma A.13}\end{aligned}$$

As D_k is an I-domain, we have $F(D) \cap \downarrow a = F(D_k) \cap \downarrow a \Subset D_k$, which implies that D is an I-domain. ■

I-domains enjoy the attractive property that every projection is automatically finitary.

A.25 Lemma *Let D be an I-domain and let $\pi : D \to D$ be a projection. Then π is finitary.*

Proof By Lemma A.13 it suffices to show that $\mathbf{im}\,\pi$ is algebraic. Since $\pi(F(D)) \subseteq F(D)$ by Axiom I, we have $\pi(F(D)) \subseteq F(D) \cap \mathbf{im}\,\pi$. Algebraicity follows using a simple calculation.

$$v = \pi(v) = \pi(\bigsqcup F(D) \cap \downarrow v) = \bigsqcup \pi(F(D) \cap \downarrow v) \sqsubseteq \bigsqcup F(\mathbf{im}\,\pi) \cap \downarrow v \sqsubseteq v$$ ■

I-domains are closely related to stable functions which were introduced by Berry [20] in an attempt to extend the property of sequentiality respectively approximations thereof to higher types.

A.26 Definition *A function* $\varphi : D \to E$ *is called* stable *iff* $\{v \sqsubseteq x \mid y \sqsubseteq \varphi(v)\}$ *has a least element, for all* $x \in D$ *and* $y \in E$*, such that* $y \sqsubseteq \varphi\, x$*. The least element is written as* $m(\varphi, x, y)$*. The set of all stable functions from* D *to* E *is denoted by* $D \to_s E$*. The stable order* $\sqsubseteq_s$ *is defined by, for stable* $\varphi_1, \varphi_2 : D \to_s E$*,*

$$\varphi_1 \sqsubseteq_s \varphi_2 \iff \varphi_1 \sqsubseteq \varphi_2 \wedge (\forall x \in D)(\forall y \in E)\; y \sqsubseteq \varphi_1(x) \Longrightarrow m(\varphi_1, x, y) = m(\varphi_2, x, y).$$

The following fact establishes the connection between stable functions and consistently multiplicative functions on I-domains.

A.27 Fact *Let* D *be an* I*-domain and let* $\varphi, \varphi_1, \varphi_2 : D \to_s E$ *be stable functions. Then*

1. φ *is stable if and only if*
$$(\forall x_1, x_2 \in D)\ \{x_1, x_2\} \text{ bounded} \Longrightarrow \varphi(x_1 \sqcap x_2) = \varphi(x_1) \sqcap \varphi(x_2). \tag{A.2}$$
2. $\varphi_1 \sqsubseteq_s \varphi_2$ *if and only if* $\varphi_1 \sqsubseteq \varphi_2$ *and*
$$(\forall x_1, x_2 \in D)\ \{x_1, x_2\} \text{ bounded} \Longrightarrow \varphi_1(x_1) \sqcap \varphi_2(x_2) = \varphi_1(x_2) \sqcap \varphi_2(x_1). \tag{A.3}$$

The set of stable functions, ordered by $\sqsubseteq_s$, is a cpo.

A.28 Lemma *Let* D *be an* I*-domain and let* $\Phi \subseteq D \to_s E$ *be a set of stable functions which is directed with respect to* $\sqsubseteq_s$*. Then* $\bigsqcup_s \Phi$ *exists and equals* $\bigsqcup \Phi$*. Hence* $\langle D \to_s E; \sqsubseteq_s\rangle$ *is a cpo.*

Proof We first show that $\bigsqcup \Phi$ is stable using property (A.2). Note that Φ is also directed wrt $\sqsubseteq$, since $\varphi_1 \sqsubseteq_s \varphi_2$ implies $\varphi_1 \sqsubseteq \varphi_2$. Let $x_1, x_2 \in D$ and assume that x_1 and x_2 are bounded.

$$\begin{array}{lll}
& (\bigsqcup \Phi)(x_1 \sqcap x_2) & \\
= & \bigsqcup\{\, \varphi(x_1 \sqcap x_2) \mid \varphi \in \Phi \,\} & \text{join is formed pointwise} \\
= & \bigsqcup\{\, \varphi(x_1) \sqcap \varphi(x_2) \mid \varphi \in \Phi \,\} & \text{by (A.2)} \\
= & \bigsqcup\{\, \varphi_1(x_1) \sqcap \varphi_2(x_2) \mid \varphi_1 \in \Phi \wedge \varphi_2 \in \Phi \,\} & \Phi \text{ is directed} \\
= & \bigsqcup\{\, \varphi_1(x_1) \mid \varphi_1 \in \Phi \,\} \sqcap \bigsqcup\{\, \varphi_2(x_2) \mid \varphi_2 \in \Phi \,\} & \sqcap \text{ is continuous} \\
= & (\bigsqcup \Phi)(x_1) \sqcap (\bigsqcup \Phi)(x_2) & \text{join is formed pointwise}
\end{array}$$

Furthermore, $\varphi \sqsubseteq_s \bigsqcup \Phi$, for all $\varphi \in \Phi$, as the following calculation based on Property (A.3) shows.

$$\begin{array}{lll}
& \varphi(x_1) \sqcap (\bigsqcup \Phi)(x_2) & \\
= & \varphi(x_1) \sqcap \bigsqcup\{\, \bar\varphi(x_2) \mid \bar\varphi \in \Phi \,\} & \text{join is formed pointwise} \\
= & \bigsqcup\{\, \varphi(x_1) \sqcap \bar\varphi(x_2) \mid \bar\varphi \in \Phi \,\} & \sqcap \text{ is continuous} \\
= & \bigsqcup\{\, \varphi(x_1) \sqcap \bar\varphi(x_2) \mid \bar\varphi \in \Phi \wedge \varphi \sqsubseteq_s \bar\varphi \,\} & \Phi \text{ is directed} \\
= & \bigsqcup\{\, \varphi(x_2) \sqcap \bar\varphi(x_1) \mid \bar\varphi \in \Phi \wedge \varphi \sqsubseteq_s \bar\varphi \,\} & \text{by (A.3)} \\
= & \bigsqcup\{\, \varphi(x_2) \sqcap \bar\varphi(x_1) \mid \bar\varphi \in \Phi \,\} & \Phi \text{ is directed} \\
= & \varphi(x_2) \sqcap \bigsqcup\{\, \bar\varphi(x_1) \mid \bar\varphi \in \Phi \,\} & \sqcap \text{ is continuous} \\
= & \varphi(x_2) \sqcap (\bigsqcup \Phi)(x_1) & \text{join is formed pointwise}
\end{array}$$

It follows that $\bigsqcup \Phi$ is the least upper bound of Φ with respect to $\sqsubseteq_s$. ∎

The following lemma is useful for identifying functionals as continuous with respect to the stable ordering.

A.29 Lemma *Let D and F be I-domains, let $\psi : (D \to_s E) \to (F \to_s G)$ be continuous wrt $\sqsubseteq$. Then ψ is monotone wrt $\sqsubseteq_s$ if and only if ψ is continuous wrt $\sqsubseteq_s$.*

Proof '$\Longleftarrow$': Trivial. '$\Longrightarrow$': Let $\Phi \subseteq D \to_s E$ be directed wrt $\sqsubseteq_s$. By Lemma A.28 we know that $\bigsqcup_s \Phi = \bigsqcup \Phi$. Furthermore, $\psi(\Phi)$ is also directed wrt $\sqsubseteq_s$, since ψ is monotone wrt $\sqsubseteq_s$. Hence $\bigsqcup_s \psi(\Phi) = \bigsqcup \psi(\Phi)$ and we have

$$\psi(\textstyle\bigsqcup_s \Phi) = \psi(\bigsqcup \Phi) = \bigsqcup \psi(\Phi) = \bigsqcup_s \psi(\Phi). \qquad \blacksquare$$

Finally, stable functions are closed under composition.

A.30 Lemma *Let $\psi : D \to E$ and $\varphi : E \to F$ be stable functions. Then $\varphi \circ \psi$ is stable and for $x \in D$, $y \in F$ with $y \sqsubseteq (\varphi \circ \psi)(x)$,*

$$m(\varphi \circ \psi, x, y) \quad = \quad m(\psi, x, m(\varphi, \psi\, x, y)). \tag{A.4}$$

Furthermore the composition is monotone with respect to $\sqsubseteq_s$.

Proof Note that $y' = m(\varphi, \psi\, x, y)$ and $v = m(\psi, x, y')$ are well-defined. We show that $v \sqsubseteq x$ satisfies $y \sqsubseteq (\varphi \circ \psi)(v)$ and that v is the least such value. Using $y \sqsubseteq \varphi\, y'$ and $y' \sqsubseteq \psi\, v$ we have $y \sqsubseteq \varphi(\psi\, v)$. Assume that $u \sqsubseteq x$ and $y \sqsubseteq (\varphi \circ \psi)(u)$. Minimality of y' implies $y' \sqsubseteq \psi\, u$ and consequently $u \sqsubseteq v$ by minimality of v.

To see that the composition is monotone wrt $\sqsubseteq_s$, let $\varphi_1 \sqsubseteq_s \varphi_1'$ and $\varphi_2 \sqsubseteq_s \varphi_2'$.

$$\begin{aligned} m(\varphi_2, x, m(\varphi_1, \varphi_2\, x, y)) &= m(\varphi_2, x, m(\varphi_1', \varphi_2\, x, y)) && \text{by } \varphi_1 \sqsubseteq_s \varphi_1' \\ &= m(\varphi_2, x, m(\varphi_1', \varphi_2'\, x, y)) \\ &= m(\varphi_2', x, m(\varphi_1', \varphi_2'\, x, y)) && \text{by } \varphi_2 \sqsubseteq_s \varphi_2' \ \blacksquare \end{aligned}$$

Appendix B

Source Code

Few things are harder to put up with than the annoyance of a good example.
— Mark Twain, *Pudd'nhead Wilson (1894) ch. 19*

The primary purpose of the DATA *statement is to give names to constants; instead of referring to π as 3.141592653589793 at every appearance, the variable* PI *can be given that value with a* DATA *statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of π change.*
— FORTRAN *manual for Xerox computers*

This appendix provides the source code of all examples used in this work. On the whole we stick to the syntax introduced in Section 2.2. Minor exceptions include the use of infix notation for system-defined functions, major exceptions such as the use of list comprehensions, irrefutable patterns, and local function definitions are explained as we proceed.

The majority of the functions is probably well-known to everyone with some experience in functional programming. There are, however, some functions or functional algorithms to which we would like to draw the attention of the reader. One is a bottom-up variant of mergesort whose definition is particularly simple and which sorts at a great pace (it beats top-down mergesort which is included in the standard environment of MIRANDA by a factor of two). Another is an adaptation of heapsort which works directly on balanced binary heaps rather than on their array representation. Since the definitions serve as examples for the context-based strictness analysis, we have included two more substantial programs as to show that the analysis, especially the generic analysis, can also cope with non-toy examples.

The sign $\Rightarrow$ which is used from time to time means 'reduces to' and refers to the operational semantics of the language. We will sometimes be sloppy and mix syntax and semantics. The bottom element $\perp$, for example, is used within expressions and represents an expression which does not have a head normalform, for example, `undef` defined by `undef = undef`.

Plan of the chapter Section B.1 and Section B.2 introduce elementary functions on truth values and on pairs, respectively. Section B.3 contains numerous list-processing functions including two variants of mergesort and quicksort. Functions on binary trees including two variants of heapsort are discussed in Section B.4. A propositional theorem prover based on the sequent calculus is developed at some length in Section B.5. Finally, Section B.6 presents a pretty-printer for the first-order language used throughout this dissertation.

B.1 Functions on truth values

The data type of truth values is one of the simplest examples for an enumeration type.

```
bool ::= False | True
```

Since the language under consideration is lazy the conditional if · then · else · need not be primitive. It may be defined via a simple case analysis on its Boolean argument. The definition of new control structures is one of the advantages of lazy languages.

```
if :: bool α α → α
if x y z = case x of True → y | False → z
```

[Nonetheless, we will treat `if` as system-defined for reasons explained in Section 7.6.1]. The 'sequential or' whose denotation was introduced in Section A.3 is equally easy formulated.

```
sor :: bool bool → bool
sor a b = case a of True → True | False → b
```

[Yes, `sor` could have been defined by `sor a b = if a True b`.] The name sequential or is due to the asymmetric behaviour of the function: `sor True` $\perp \Rightarrow$ `True` but `sor` $\perp$ `True` $\Rightarrow$ $\perp$. Note that the 'parallel or' cannot be user-defined since the language like most functional languages is inherently sequential.

B.2 Functions on pairs

Polymorphic pairs may be introduced by a data type definition consisting of a single constructor.

```
(α1,α2) ::= (α1,α2)
```

Note that we use the mixfix notation `(·,·)` both for types and values. A definition which adheres more closely to the syntax is `pair` $\alpha_1\,\alpha_2$ `::= Pair` $\alpha_1\,\alpha_2$. The following definitions introduce some simple functions working on polymorphic pairs.

```
fst :: (α1,α2) → α1
fst x = case x of (a,b) → a

snd :: (α1,α2) → α1
snd x = case x of (a,b) → b

dup :: α → (α,α)
dup a = (a,a)
```

B.3 Functions on lists

The data type of lists is pervasive in every functional programming language. Their common ancestor LISP, which is an acronym for list processing language, is even named after them. Polymorphic lists are introduced by the following definition.

```
[α] ::= [] | α:[α]
```

It is common practice to use similar syntax for both types and values: `[num]` denotes the type of all lists with numerical elements while `[1,2,3]` serves as a shorthand for `1:2:3:[]`. The list constructor '`:`' is pronounced 'cons' (the name traces back to LISP) while the empty list `[]` is often called 'nil' (nil is a contraction of the Latin word nihil which means nothing).

Lists may be conceptionally infinite. The infinite list of all integers from n upwards is generated by a call to `from`.

```
from :: num → [num]
from n = n:from (n+1)
```

When reasoning about lists and explaining functions on lists it is therefore necessary to consider the possibility of infinite and partial lists (a list ending in $\perp$ is called partial).

List concatenation is a simple example for a recursive definition over lists. Note that the case analysis is essentially dictated by the data type definition.

```
append :: [α] [α] → [α]
append x y = case x of [] → y |
                       a:w → a:append w y
```

The effect of `append` is clear if a finite list is provided as the first argument, but what happens if the list is partial or infinite? Well, then the result is simply the first argument, that is, `append` $x\ y = x$.

The running time of `append` is proportional to the length of its first argument. This fact has sometimes unwelcome consequences. Consider, for example, the following implementation of the function `reverse`, which reverses the order of elements of a (finite) list.

```
reverse :: [α] → [α]
reverse x = case x of [] → [] |
                      a:w → append (reverse w) [a]
```

The repeated use of `append` results in a running time proportional to n^2 where n is the length of the argument list. It is not unjust to say that part of the folklore of functional programming is concerned with the problem of eliminating calls to `append`. An efficient variant of reverse is constructed as follows: Define an auxiliary function `revaux` which satisfies `revaux` $x\ y =$ `append (reverse` x`)` y, that is, solve the harder problem of reversing a list and appending a second list.

```
revaux :: [α] [α] → [α]
revaux x y = case x of [] → y |
                       a:w → revaux w (a:y)
```

We obtain a linear variant of list reversal by setting `reverse` $x =$ `revaux` x `[]`. It is an easy exercise to prove both definitions equivalent.

The function definitions given so far are based on an exhaustive case analysis. While this is certainly a desirable property it can sometimes not be met. The function `head` which extracts the first element of a non-empty list serves as an example.

```
head :: [α] → α
head x = case x of a:x → a
```

An incomplete case analysis generally gives rise to a partial function: `head`, for example, is undefined on the empty list.

The functions `sum`, `length`, and `and` are typical examples for list consuming functions. They take a list and produce a scalar value. [Note the striking similarity of their definitions which is explained in Section 7.6.3.]

```
sum :: [num] → num
sum x = case x of [] → 0 |
                  a:w → a+sum w


length :: [α] → num
length x = case x of [] → 0 |
                     a:w → 1+length w


and :: [bool] → bool
and x = case x of [] → True |
                  a:w → a & and w
```

The function sum calculates the sum of list of numerical values, length determines the length of a list, and and forms the logical conjunction of a list of Boolean values.

The same recursive scheme is employed in the definition of concat which concatenates a list of lists into a single list.

```
concat :: [[α]] → [α]
concat xs = case xs of [] → [] |
                       x:ws → append x (concat ws)
```

The function member takes a list and a value and checks whether the value is present in the list. [Note that the list comes first. Note, too, that member is restricted to numerical lists. This is due to the use of = which has the type num num → num.]

```
member :: [num] num → bool
member x a = case x of [] → False |
                       b:w → a=b \/ member w a
```

Lists may be manipulated using drop and take. Both take a number and a list; drop removes that many elements from the front of the list while take returns the specified number of elements.

```
drop :: num [α] → [α]
drop n x = if n=0 then x
           else case x of [] → [] |
                          a:w → drop (n-1) w


take :: num [α] → [α]
take n x = if n=0 then []
           else case x of [] → [] |
                          a:w → a:take (n-1) w
```

Note that the functions drop and take have been defined in such a way that they satisfy the equation append (drop n x) (drop n x) $= x$, for all natural numbers n and all lists x.

Top-down mergesort All in all we will consider three different sorting algorithms. We start with an implementation of mergesort which is found, for example, in the standard environment of MIRANDA. Top-down mergesort is a typical instance of the divide and conquer scheme. Divide step: If the length of the argument list is less or equal one, it is trivially sorted. Otherwise the argument is divided into two segments[1] of equal length using `take` and `drop`. Both segments are then sorted via recursive calls. Conquer step: The resulting sorted lists are interleaved using `merge`.

```
sort :: [num] → [num]
sort x = let n = length x in
         if n<=1 then x
         else let n2 = n div 2 in
              merge (sort (take n2 x)) (sort (drop n2 x))
```

The `merge` function takes two sorted lists and merges them to produce a single sorted list. It proceeds by nested case analysis on the first and second argument.

```
merge :: [num] [num] → [num]
merge x y = case x of
            [] → y |
            a:v → case y of
                  [] → a:v |      // note: a:v instead of x
                  b:w → if a<=b then a:merge v (b:w)
                                else b:merge (a:v) w
```

Top-down mergesort is an optimal sorting algorithm since it exhibits $O(n \log n)$ worst-case behaviour. However, let us take a closer look at the implementation to determine the constants hidden in the big-oh notation. An adequate cost model in based on the number of '`:`' operations rather than on the number of comparisons. Now, the divide phase contributes $\frac{n}{2} \log n$ '`:`' operations since `take` n performs n cons'es while `drop` n comes for free. The call `merge` x y costs at most $m + n$ where m is the length of x and n the length of y. This makes up for a total of $n \log n$ for the conquer phase. All in all we have $\frac{3}{2} n \log n$ '`:`' operations.

Note that the divide phase is rather unproductive as indicated by the absence of comparisons operations (hence it is unwise to base the run-time analysis on the number of comparisons). Bottom-up mergesort improves on this as explained below.

Bottom-up mergesort The bottom-up variant of mergesort originates in algorithms which have been developed for sorting external data. It is typically assumed that the data resides on magnetic tapes whose access characteristics are akin to that of lists. External sorting techniques are based on the concept of a *run* which is a sorted segment of a file. In an initial phase the original file is first divided into runs. Then the file is repeatedly traversed thereby merging runs until only one run is left. [This is usually accomplished using several magnetic tapes but do not let us go into details here.]

We mimic this technique by using lists of lists, the element lists corresponding to runs. The initial list of runs is constructed simply by dividing the given list into singletons lists. [One can, of course, do better.]

```
msort :: [num] → [num]
msort x = loop [ [a] | a<-x ]
```

[1] A list x is said to be a segment of y if there exist lists v and w such that `append (append` v x`)` $w = y$.

If the list of runs is a singleton we are finished, otherwise at least one additional pass is required.

```
loop :: [[num]] → [num]
loop xs = case xs of
          [] → [] |
          x1:xs1 → case xs1 of
                   [] → x1 |
                   x2:xs2 → loop (merge x1 x2:pass xs2)
```

The function pass draws repeatedly two runs from its input list merging them into a single list until the input list is empty. A single pass halves the length of the outer list and doubles the length of the runs.

```
pass :: [[num]] → [[num]]
pass xs = case xs of
          [] → [] |
          x1:xs1 → case xs1 of
                   [] → [x1] |
                   x2:xs2 → merge x1 x2:pass xs2
```

It is immediate that bottom-up mergesort has $\mathrm{O}(n \log n)$ worst-case behaviour. But how does the number of ':' operations compare to the top-down variant? The costs of the calls to merge are, of course, the same: They amount to $n \log n$. Additionally we have to consider the cons'es for the outer lists which run up to $\frac{n}{2} + \frac{n}{4} + \cdots + 1 = n - 1$ operations. Including the initialization we have at most $3n + n \log n$ operations which compares favourably with top-down mergesort.

Note that the outer as well as the inner lists are never constructed in their entirety due to the lazy evaluation strategy. In a strict language additional bookkeeping is required to achieve this effect. An ingenious method termed 'smooth applicative mergesort' due to O'Keefe [127] exhibits a similar run-time behaviour.

Technical remark: The definition of msort above employs a list comprehension of the form [e_1 | a<-e_2] which may be easily translated to the source language—provided e_1 contains no free variables except a. The list comprehension is replaced by the call map e_2 where map is a fresh function symbol. The global function map is then defined by

```
map :: [σ1] → [σ2]
map x = case x of [] → [] |
                  a:w → e1:map w.
```

Quicksort The quicksort algorithm was invented and named by C.A.R. Hoare. It ranks among the best sorting algorithms for arrays. However, as matters stand the list-based variant presented below is rather slow. [We have included it nonetheless as sorting algorithms serve primarily as examples for strictness analysis.]

```
qsort :: [num] → [num]
qsort x = case x of
            [] → [] |
            a:w → let (x1,x2) = partition a w in
                  append (qsort x1) (a:qsort x2)
```

```
partition :: num [num] → ([num],[num])
partition a x = case x of
                  [] → ([],[]) |
                  b:w → let (x1,x2) = partition a w in
                        if b<=a then (b:x1,x2)
                                else (x1,b:x2)
```

Technical remark: Both qsort and partition use pairs of variables on the left hand side of local definitions. Again, the syntactic sugar may be removed via a simple transformation: The pattern (x_1, x_2) is replaced by a fresh variable, say, x and occurrences of x_i are replaced by case x of $(x_1, x_2) \rightarrow x_i$ respectively. This technique generalizes to arbitrarily nested patterns consisting of tuple constructors and variables (so called irrefutable patterns). It furthermore applies to every variable binding construct: function definitions, case analysis, local definitions and list comprehensions.

The quicksort variant above suffers from two shortcomings. Firstly, the partition of the list is rather costly due to the heavy use of pairs. Secondly, the calls to append contribute O(n^2) cons operations in the worst-case. The following variant due to Paulson [128] improves on this using accumulators (we have qsort x = qsortaux x []).

```
qsortaux :: [num] [num] → [num]
qsortaux x y
    = case x of
      [] → [] |
      a:v → let partition l r z
                  = case z of
                    [] → qsortaux l (a:qsortaux r y) |
                    b:w → if b<=a then partition (b:l) r w
                                  else partition l (b:r) w
            in partition [] [] v
```

Technical remark: The definition of qsortaux involves a local function definition. We may globalize this definition by making the free variables of partition, y and a, into extra parameters.

```
qsortaux :: [num] [num] → [num]
qsortaux x y
    = case x of [] → [] |
                a:v → partition y a [] [] v

partition :: [num] num [num] [num] [num] → [num]
partition y a l r z
    = case z of [] → qsortaux l (a:qsortaux r y) |
                b:w → if b<=a then partition y a (b:l) r w
                              else partition y a l (b:r) w
```

This transformation known as λ-lifting [95] stems from work about the implementation of functional programming languages. Abstract machines like the G-machine [132] or the Three-Instruction-Machine [56, 132] are not equipped with a mechanism to deal with environments. Therefore the compilation process requires an additional λ-lifting pass which transforms a given program so that there are no local function definitions.

Primes The conceptional advantage of working with infinite lists is nicely demonstrated by the function `primes`, which generates the list of all prime numbers, and its subsidiaries.

```
primes ::  ε → [num]
primes = 2:sieve (from 3) primes;

sieve :: [num] [num] → [num]
sieve x r = case x of a:w → if fact a r then sieve w r
                                        else a:sieve w r

fact :: num [num] → bool
fact i x = case x of j:w → j*j<=i & (i mod j=0 \/ fact i x)
```

The function `sieve` x r removes all numbers from x which are divisible by some number of r. Note that the auxiliary `fact` stops testing for divisibility if the square of a factor exceeds the number to be tested (therefore the list of factors must be in ascending order). It is immediate that `sieve` works the faster the smaller the list of factors is. Therefore the list of primes is employed in its own generation! To base the recursive definition of `primes` the first prime, that is, 2 must be stated explicitly.

Note that both `sieve` and `fact` are deliberately designed for infinite lists as they do not consider the case of the empty list.

B.4 Functions on trees

The type of labelled binary trees is given by the subsequent data type definition.

```
tree α ::= Empty | Node (tree α) α (tree α)
```

The equation is paraphrased as follows: A tree is either the empty tree or a labelled node with a left and a right subtree.

A simple example for a recursive definition over trees is the function `size` which counts the number of nodes in a given tree.

```
size :: tree α → num
size t = case t of Empty → 0 |
                   Node l a r → size l + 1 + size r
```

The function `size` plays the same rôle as `length` does for lists.

Inorder traversal The function `inorder` converts a tree to a list such that the elements of the list are exactly the elements of the tree. The relative order of the elements is as follows: The label of each node lies between the labels of its left and its right subtree (whence the name of the function). A naïve definition of `inorder` is given below.

```
inorder :: tree α → [α]
inorder t = case t of
            Empty → [] |
            Node l a r → append (inorder l) (a:inorder r)
```

Due to the use of append the running time is in the worst-case proportional to n^2 where n is the size of the input tree as defined above.[2]

Using a similar approach as in the case of `reverse` we obtain again a linear variant. First introduce an auxiliary function `inaux` which satisfies `inaux` $t\ x =$ `append (inorder` t`)` x and then set `inorder` $t =$ `inaux` t `[]`.

```
inaux :: (tree α) [α] → [α]
inaux t x = case t of Empty → x |
                      Node l a r → inaux l (a:inaux r x)
```

An alternative linear definition is based on the following idea: The call to append is unnecessary if the left subtree is empty. We work towards this simple case by repeatedly rotating the given tree to the right. A rotation to the right leaves the order of elements unchanged while it decreases the height of the left subtree by one.

```
inorder :: tree α → [α]
inorder t = case t of
            Empty → [] |
            Node l b r → case l of
                         Empty → b:inorder3 r |
                         Node ll a lr →
                             inorder3 (Node ll a (Node lr b r))
```

Preorder traversal Preorder traversal differs from inorder traversal only in the order of elements: The label of each node is put before the labels of the left and right subtrees. Consequently the definitions of `inorder` and `inaux` need only minor modifications to realize a preorder traversal.

```
preorder :: tree α → [α]
preorder t = case t of
             Empty → [] |
             Node l a r → a:append (preorder l) (preorder r)

preaux :: (tree α) [α] → [α]
preaux t x = case t of
                 Empty → x |
                 Node l a r → a:preaux l (preaux r x)
```

An alternative implementation is based on the function `flatten` which uses a stack of trees for bookkeeping purposes. Using `concat` which concatenates a list of lists into a single list `flatten` may be specified by `flatten` $ts =$ `concat [ preorder` t `|` t`<-`ts `]`.

[2] The definition of `inorder` suffers from the same problems as the first definition of `reverse` (cf to Section B.3). To see the connection consider the function `lefty` given by

```
lefty :: [α] → tree α
lefty x = case x of [] → Empty |
                    a:w → Node (lefty w) a Empty.
```

The function `lefty` maps a list to a (degenerated) tree whose subtrees are always empty. Now, `reverse` and `inorder` are related by `reverse` $x =$ `inorder (lefty` x`)`. Moreover, the evaluation of both expressions differs only in a constant factor.

```
flatten :: [tree α] → [α]
flatten ts = case ts of
               [] → [] |
               t:ts’ → case t of
                         Empty → flatten ts’ |
                         Node l a r → a:flatten (l:r:ts’)
```

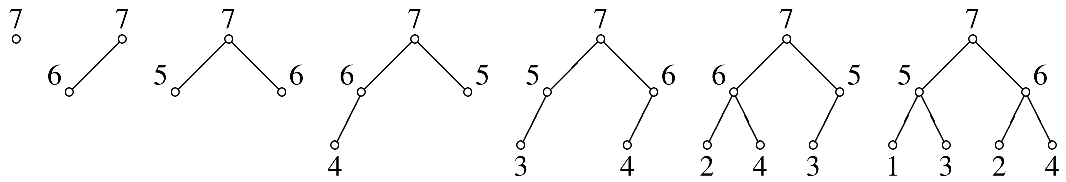
Top-down heapsort Heapsort was named after the central data structure it employs: A heap is a binary tree such that the label of each node is less or equal than the labels of its left and right subtrees. Consequently, the least element is contained in the root of a heap. The heapsort algorithm consists of two phases: the construction of the heap and the reduction of a heap into a sorted list. The following top-down variant is due to the author, see [73].

```
hsort :: [num] → [num]
hsort x = unheap (build x)

build :: [num] → tree num
build x = case x of [] → Empty |
                    a:w -> sink a (build w)
```

The function `sink` inserts an element into a heap. The heaps created by repeated calls to `sink` are (size-) balanced, that is, the size of the left and right subtrees of each node differ by at most one. This remarkable property is established along the following lines: First note that the order of the subtrees plays virtually no rôle. A heap may be mirrored ad libitum without compromising the heap property. Therefore one has a choice when inserting an element as to whether the left or the right subtree is considered next. This choice is put to clever use by `sink` which always inserts the element into the right subtree but subsequently exchanges the left and the right subtree so that the next element sinks into the (formerly) left subtree.

```
sink :: num (tree num) → tree num
sink a t = case t of
             Empty → Node Empty a Empty |
             Node l b r → if a<=b then Node (sink b r) a l
                                  else Node (sink a r) b l
```

The effect of sinking the elements 7, 6, . . . , 1 into the empty heap is pictured below. Note that `sink` produces a complete binary heap.



The second phase is more or less a matter of routine. The function `unheap` extracts the root element of the heap, calls `reheap` which combines the left and the right subheap to form a single heap, extracts the new root element and so forth.

```
unheap :: tree num → [num]
unheap t = case t of Empty → [] |
                     Node l a r → a:unheap (reheap l r)
```

```
reheap :: (tree num) (tree num) → tree num
reheap t u
    = case t of
      Empty → u |
      Node l a r →
          case u of
          Empty → Node l a r |
          Node rl b rr →
              if a<=b then Node (reheap l r) a (Node rl b rr)
                      else Node (Node l a r) b (reheap rl rr)
```

Since the depth of the heap is bounded by $\lceil \log n \rceil$ heapsort has a $O(n \log n)$ worst-case behaviour.

Bottom-up heapsort It is well-known that is more efficient to construct the heap bottom-up rather than top-down. The bottom-up variant works as follows. Assume that a complete binary *tree* is given. Working up from the leaves towards the root the heap property is established levelwise by repeatedly turning an inner node and two subheaps to a heap. The definition of the last operation is straightforward.

```
heapify :: num (tree num) (tree num) → tree num
heapify a l r
= case l of
  Empty → (case r of
           Empty → Node Empty a Empty |
           Node rl c rr → swapr Empty a rl c rr) |
  Node ll b lr → case r of
                 Empty → swapl ll b lr a Empty |
                 Node rl c rr → if b<=c then swapl ll b lr a r
                                        else swapr l a rl c rr

swapl :: (tree num) num (tree num) num (tree num) → tree num
swapl ll b lr a r = if a<=b then Node (Node ll b lr) a r
                            else Node (heapify a ll lr) b r

swapr :: (tree num) num (tree num) num (tree num) → tree num
swapr l a rl b rr = if a<=b then Node l a (Node rl b rr)
                            else Node l b (heapify a rl rr)
```

For reasons of efficiency the bottom-up construction of the heap works directly on a given list rather than on a complete binary heap built beforehand.

```
buildn :: num [num] → (tree num,[num])
buildn n x = if n=0 then (Empty,x)
             else case x of
                  [] → (Empty,[]) |
                  a:u → let n2 = (n-1) div 2;
                                (l,v) = buildn n2 u;
                                (r,w) = buildn (n-1-n2) v
                             in (heapify a l r,w)
```

The running time of `buildn` is proportional to n where n is the length of the argument list which compares favourably to `build` which takes $\Theta(n \log n)$ steps. Finally, bottom-up heap-sort is given by.

```
bhsort :: [num] → [num]
bhsort x = unheap (fst (buildn (length x) x))
```

B.5 A propositional theorem prover

This section introduces a theorem prover for propositional logic which is inspired by the far more ambitious tactical theorem prover in [128]. The term 'theorem prover' is slightly exaggerated, 'tautology checker' is probably more appropriate as the main function, called `taut`, takes a propositional formula and checks whether it is a tautology (a formula is a tautology iff it is true for all assignments of truth values to propositional variables).

A propositional formula is build up from propositional variables (or atoms) using the logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$. They are represented by the following data type (note that propositional variables are simply represented by numbers).

```
form ::= Atom num                    // propositional variable
       | Not form                    // negation
       | And form form               // conjunction
       | Or form form                // disjunction
       | Then form form              // implication
       | Iff form form               // equivalence
```

[It should be noted that the type `form` also encompasses infinite formulae such as `nats 0` defined by `nats n = Or (Atom n) (nats (n+1))`. This contrasts to the common inductive definition of propositional formulae which admits only finite terms.]

The tautology checker is based on the sequent calculus which originates in papers by Gentzen, see [60, 128]. In testing whether a given formula is a tautology the sequent calculus systematically tries to construct a counterexample, that is, an assignment which falsifies the formula. If the search for a counterexample fails we may conclude that the given formula is indeed a tautology. For example, in order to falsify the formula $a \rightarrow (b \rightarrow a)$ we must make a true and $b \rightarrow a$ false. Now, in order to make $b \rightarrow a$ false we must make b true and a false. Since the atom a cannot be made true and false at the same time, it is not possible to construct a counterexample which in turn implies that $a \rightarrow (b \rightarrow a)$ is tautological. These ideas lead naturally to the notion of a sequent which has the form

$$\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n,$$

where $\phi_1, \ldots, \phi_m$ and $\psi_1, \ldots, \psi_n$ are multisets of formulae (called antecedent and succedent). Semantically, the sequent has the same meaning as the formula $\phi_1 \wedge \ldots \wedge \phi_m \rightarrow \psi_1 \vee \ldots \vee \psi_n$. Thus in order to falsify a sequent an assignment must make the formulae $\phi_1, \ldots, \phi_m$ all true and the formulae $\psi_1, \ldots, \psi_n$ all false.

The deduction rules of the sequent calculus come in pairs: One rule deals with a connective appearing on the left (trying to make the respective formula true) and a second rule operates on a connective appearing on the right (trying to make the formula false). The two sequent rules for the implication are listed below (the premises are above the line, the conclusion below).

$$\frac{\Gamma \vdash \Delta, \phi \qquad \psi, \Gamma \vdash \Delta}{\phi \to \psi, \Gamma \vdash \Delta} \qquad\qquad \frac{\phi, \Gamma \vdash \Delta, \psi}{\Gamma \vdash \Delta, \phi \to \psi}$$

Similar rules may be stated for the remaining connectives (cf to `left` and `right` below). An atomic sequent consists only of propositional variables, that is, no deduction rule is applicable to the sequent. An axiom is an atomic sequent such that the antecedent and the succedent contain a common propositional variable.

$$a, \Gamma \vdash \Delta, a$$

Clearly, an axiom is not falsifiable. In order to check quickly whether a sequent is atomic both antecedent and succedent are further divided into $\Gamma; A \vdash \Delta; B$ where A and B are multisets of propositional variables. The multisets A and B record the atoms found during a proof. Since we represent multisets simply by lists and sets by ordered lists with no duplicates sequents and atomic sequents are represented as follows (== introduces a type abbreviation).

```
sequent == ([form],[num],[form],[num])
atomic  == ([num],[num])
```

The tautology check may be logically divided into three steps. Starting with the sequent $\vdash \phi$ a deduction tree is build by systematically applying the inference rules of the sequent calculus. The resulting deduction tree is then traversed (depth-first or breadth-first) thereby collecting the atomic sequents labelled to the leaves. Finally, axioms are deleted from the list of atomic sequents returning a list of counterexamples. Thus the formula ϕ is valid iff the list of counterexamples is empty.

Set-theoretic functions The tautology checker makes use of two auxiliary functions. The function add adds an element to a given set. Its definition is, of course, tailored to the representation of sets by ordered lists that do not contain duplicates.

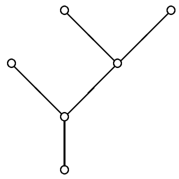
```
add :: num [num] → [num]
add a x = case x of
               [] → [a] |
               b:w → if      a<b then a:b:w
                     else if a=b then b:w
                     else             b:add a w
```

The function `intersection` computes the intersection of two given sets. Due to the set representation its running time is proportional to the maximum of the length of the two lists.

```
intersection :: [num] [num] → [num]
intersection x y
    = case x of
      [] → [] |
      a:v → case y of
            [] → [] |
            b:w → if      a<b then intersection v (b:w)
                  else if a>b then intersection (a:v) w
                  else             a:intersection v w
```

Building a deduction tree A deduction tree is build by repeatedly applying the inference rules to a given sequent. The deduction tree for $\vdash \neg(a \to (b \to a))$ has the following shape (the schematic picture on the right makes the tree structure more explicit).

$$\dfrac{\dfrac{\dfrac{}{\vdash a} \qquad \dfrac{\dfrac{}{\vdash b} \quad \dfrac{}{a \vdash}}{b \to a \vdash}}{a \to (b \to a) \vdash}}{\vdash \neg(a \to (b \to a))}$$



The tree is actually a counterexample tree since the sequents labelled to the leaves are falsifiable. Accordingly, a tree whose leaves are labelled with axioms is a proof tree. Here is the proof tree for $\vdash a \to (b \to a)$.

$$\dfrac{\dfrac{\dfrac{}{a, b \vdash a}}{a \vdash b \to a}}{\vdash a \to (b \to a)}$$

Labelled trees are introduced by the following data type definition. Since we are only interested in atomic sequents we decided not to attach labels to the inner nodes.

```
dtree α ::= Leaf α                    // leaf node
          | Inner [dtree α]           // inner node
```

The function `build` takes a sequent and builds a deduction tree the leaves of which are labelled with atomic sequents. This is trivial if the given sequent $\Gamma; A \vdash \Delta; B$ is finished which means that both Γ and Δ are empty. Otherwise, $\Gamma; A \vdash \Delta; B$ is expanded using a suitable inference rule. The strategy applied is very simple: If Γ is non-empty, then the first formula of Γ is considered. Otherwise, the first formula of Δ is taken.

```
build :: sequent → dtree atomic
build s = if finished s then Leaf (extract s)
                         else Inner [ build s' | s'<-expand s ]

finished :: ([form],[num],[form],[num]) → bool
finished (ante,a,succ,b) = empty ante & empty succ

empty :: [α] →bool
empty x = case x of [] → True | a:w → False

extract :: ([form],[num],[form],[num]) → ([num],[num])
extract (ante,a,succ,b) = (a,b)

expand :: sequent → [sequent]
expand (ante,a,succ,b) =
    case ante of
         [] → (case succ of
                     q:qs → right q ante a qs b) |
         p:ps → left p ps a succ b
```

The inference rules of the sequent calculus are coded into the functions `left` and `right`. Both functions take the principal formula, that is, the proposition to which the rule is applied as the first argument and return the premises of the respective inference rule. If the principal formula is atomic, then the predicate symbol is simply added to A or to B.

```
left :: form [form] [num] [form] [num] → [sequent]
left p ante a succ b =
    case p of
      Atom v → [(ante,add v a,succ,b)] |
       Not p → [(ante,a,p:succ,b)] |
     And p q → [(p:q:ante,a,succ,b)] |
      Or p q → [(p:ante,a,succ,b), (q:ante,a,succ,b)] |
    Then p q → [(ante,a,p:succ,b), (q:ante,a,succ,b)] |
     Iff p q → [(p:q:ante,a,succ,b), (ante,a,p:q:succ,b)]

right :: form [form] [num] [form] [num] → [sequent]
right q ante a succ b =
    case q of
      Atom v → [(ante,a,succ,add v b)] |
       Not p → [(p:ante,a,succ,b)] |
     And p q → [(ante,a,p:succ,b), (ante,a,q:succ,b)] |
      Or p q → [(ante,a,p:q:succ,b)] |
    Then p q → [(p:ante,a,q:succ,b)] |
     Iff p q → [(p:ante,a,q:succ,b), (q:ante,a,p:succ,b)]
```

Traversing a deduction tree There are basically two (uninformed) strategies for traversing a (deduction) tree: depth-first or breadth-first. Depth-first is generally more efficient while breadth-first allows to generalize the tautology checker to infinite sequents. The function `dfs` takes a list of open nodes (a node is open if its immediate subtrees have not yet been considered) and returns a list of the labels contained in the trees. The open list is used in a stack-like fashion so that the subtrees of a node a considered prior to its siblings.

```
dfs :: [dtree α] → [α]
dfs sequs = case sequs of
            [] → [] |
            n:open → case n of
                     Leaf s → s:dfs open |
                     Inner ts → dfs (append ts open)
```

The function `bfs` differs from `dfs` only in the use of the open list which now serves as a queue, that is, the subtrees are appended to the end of a list so that the siblings of the current node are considered next.

```
bfs :: [dtree α] → [α]
bfs sequs = case sequs of
            [] → [] |
            n:open → case n of
                     Leaf s → s:bfs open |
                     Inner ts → bfs (append open ts)
```

The tautology checker An atomic sequent is an axiom if antecedent and succedent have a non-empty intersection.

```
axiom :: atomic → bool
axiom (a,b) = ~empty (intersection a b)
```

The function `counterexamples` takes a list of atomic sequents and returns the list of non-axioms, commonly known as counterexamples.

```
counterexamples :: [atomic] → [atomic]
counterexamples sequs = [ s | s<-sequ; ~axiom s ]
```

Technical remark: The definition of `counterexamples` contains a list comprehension of the form `[` e_1 `|` a`<-`e_2`;` e_3 `]` where the range of the variable a is restricted by the Boolean expression e_3. The list comprehension may be replaced by the call `filter` e_2 where `filter` is a new function symbol—provided both e_1 and e_3 contain no free variables except a. The function `filter` is given by the following global definition.

```
filter :: [σ1] → [σ2]
filter x = case x of [] → [] |
                     a:w → if e3 then e1:filter w
                                 else filter w
```

The tautology checker is finally implemented by the function `taut` which simply combines `build`, `dfs`, and `counterexamples`.

```
taut :: form → bool
taut p = empty (counterexamples (dfs [build ([],[],[p],[])]))
```

Disjunctive normalform The theorem prover only checks whether the list of counterexamples is empty. If we transform the non-axiom sequents in a suitable manner we obtain a procedure which converts a proposition to disjunctive normalform. Starting with the sequent $\phi \vdash$ we first determine the counterexamples. If there are none, $\phi \vdash$ is valid implying that ϕ is unsatisfiable. In this case we output $a \wedge \neg a$. Otherwise we know that a valuation falsifies $\phi \vdash$ if and only if it falsifies some leaf sequent. Applying de Morgan we have that a valuation falsifies the non-axiom sequent $a_1, \ldots, a_m \vdash b_1, \ldots, b_n$ iff it makes the conjunction

$$a_1 \wedge \ldots \wedge a_m \wedge \neg b_1 \wedge \cdots \wedge \neg b_n$$

true. Consequently the disjunctive normalform is the disjunction of these conjuncts. The function `conjunct` transforms a given sequent non-empty atomic sequent to a conjunction.

```
conjunct :: atomic → form
conjunct (a,b) = foldr1 And (append [ Atom v | v<-a ]
                                    [ Not (Atom v) | v<-b ])
```

Technical remark: As the final example of syntactic sugar `conjunct` employs the higher-order function `foldr1` which folds right over a non-empty list. [The function `foldr` which also works a non-empty lists was introduced in Section 7.6.3.] If the functional argument is given we may replace `foldr1 (`$\oplus$`)` e by the first-order call `fold` e where `fold` is given by

```
fold :: [σ1] → σ2
fold x = case x of
          a:v → case v of [] → a |
                                b:w → a ⊕ fold (b:w)
```

which amounts to partially evaluating the definition of `foldr1` with respect to the first argument.

The function `disjunct` which forms the disjunction of a given list of atomic sequents is defined in a similar manner.

```
disjunct :: [atomic] → form
disjunct sequs = case sequs of
                 [] → And (Atom 0) (Not (Atom 0)) |
                 s:rest → foldr1 Or [ conjunct s' | s'<-s:rest ]
```

To obtain the disjunctive normalform `disjunct` is simply applied to the list of atomic sequents produced by `counterexamples`.

```
dnf :: form → form
dnf p = disjunct (counterexamples (dfs [build ([p],[],[],[])]))
```

B.6 A pretty-printer for first-order expressions

A pretty-printer converts the abstract syntax tree of a formula or a program into a textual representation. Since the generated output is assumed to be readable by humans, line breaks and indentation are used to support its logical structure. The technique is best illustrated by means of a non-trivial example. Therefore we decided to present a pretty-printer for the first-order language used in this dissertation. [It is quite amusing that the pretty printer could print itself if we had a parser.] The program given below is derived from the pretty-printing routines of [132].

The presentation proceeds in two steps: First, a 'general purpose layouter' is described which accepts block structured text incorporating information about indentation and about line breaks. In a second step we show how to generate input for the layouter by recursively descending a given first-order program. The interface between the layouter and the generator is given by the data type definition below. [The use of intermediate data structures is typical of functional programs.]

```
text ::= Str [char]                   // string
       | Block [text]                 // block of text
       | Indent text                  // indented text
       | NL                           // newline
```

A piece of text is either a string, a sequence of text elements, a text element to be indented to the current column (indentation is only effective if the indented text consists of several lines) or else a line break. Note the similarity to general trees: `Str` s corresponds to a leaf labelled with s, `Block` $[t_1, \ldots, t_n]$ corresponds to an inner node with subtrees $t_1, \ldots, t_n$. Consequently, the display of a text element amounts essentially to a tree traversal. [Note that `c` stands for column and `i` for indentation.]

```
display :: text → [char]
display t = flatten 0 [(t,0)]
```

```
flatten :: num [(text,num)] → [char]
flatten c ts
    = case ts of
      [] → [] |
      (t,i):ts1 →
          case t of
          Str s → append s (flatten (c+length s) ts1) |
          Block ys → flatten c (append [ (y,i) | y<-ys ] ts1) |
          Indent x → flatten c ((x,c):ts1) |
          NL → '\n':append (spaces i) (flatten i ts1)

spaces :: num → [char]
spaces n = if n=0 then [] else ' ':spaces (n-1);
```

The function `flatten` converts a list of text elements to a string. It keeps track of the current column and the indentation of each text element in the list.

The following functions are useful in defining pretty-printers: `ppParen` parenthesizes a text element depending on the value of its Boolean argument, `ppList` takes a list of text elements and inserts a separator between adjacent text elements.

```
ppParen :: bool text → text
ppParen flag t = if flag then t else Block [Str "(",t,Str ")"]

ppList :: [text] text → text
ppList ts s = Block (insert s ts)

insert :: α [α] → [α]
insert s x = case x of
             [] → [] |
             a:v → case v of [] → [a] |
                              b:w → a:s:insert s (b:w);
```

The data type which is employed to represent first-order expressions is a direct translation of the abstract syntax listed in Table 2.3. Note that we leave the representation of identifiers open.

```
expr α ::= Var α                                // variable
         | Freeze (expr α)                      // delay evaluation
         | Unfreeze (expr α)                    // force evaluation
         | Sys α (expr α)                       // system function call
         | Con α (expr α)                       // constructed value
         | Fun α (expr α)                       // user function call
         | Sequ [expr α]                        // sequence
         | Case (expr α) [(α,[α],expr α)]       // case analysis
         | Let [α] (expr α) (expr α)            // local value definition
         | Rec [α] (expr α)                     // recursion operator
```

Building on the layouter the pretty-printer is relatively easy to define. It uses three precedence levels to avoid redundant parenthesis.

Level 0: Non-atomic expressions must be enclosed in parenthesis.

Level 1: case-expressions must be enclosed in parenthesis.

Level 2: No expression must be enclosed in parenthesis.

Note that case-expressions require a special treatment since nested case-expressions must be bracketed to avoid ambiguity. Consider, for example, the expression

$$\underline{\texttt{case}}\ e_1\ \underline{\texttt{of}}\ \texttt{[]} \rightarrow \underline{\texttt{case}}\ e_2\ \underline{\texttt{of}}\ \texttt{[]} \rightarrow e_3 \mid a{:}w \rightarrow e_4.$$

It is not clear whether the last branch belongs to the first or to the second case. [This is an instance of the dangling else problem.]

The function showexpr converts an expression where identifiers are represented by strings into its textual representation. It is based on the auxiliary function pp which pretty prints an expression according to a given precedence level.

```
showexpr :: expr [char] → [char]
showexpr e = display (pp 2 e)

pp :: num (expr [char]) → text
pp d e
= case e of
  Var s → Str s |
  Freeze e → ppParen (d>=1) (Block [Str "freeze ",pp 0 e]) |
  Unfreeze e → ppParen (d>=1) (Block [Str "unfreeze ",pp 0 e]) |
  Sys s e → ppParen (d>=1) (Block [Str s,Str " ",pp 0 e]) |
  Con s e → ppParen (d>=1) (Block [Str s,Str " ",pp 0 e]) |
  Fun s e → ppParen (d>=1) (Block [Str s,Str " ",pp 0 e]) |
  Sequ es → ppList [ pp 0 e | e<-es ] (Str " ") |
  Case e cases → ppParen (d>=2)
                (Block [Str "case ",pp 2 e,Str " of",NL,
                        Str "     ",Indent (ppCases cases)]) |
  Let vs e e' → ppParen (d>=1)
               (Block [Str "let ",ppVars vs,Str " = ",pp 2 e,NL,
                       Str "in  ",Indent (pp 2 e')]) |
  Rec vs e → ppParen (d>=1)
            (Block [Str "rec ",ppVars vs,Str ".",pp 2 e]);

ppVars :: [[char]] → text
ppVars vs = ppList [ Str v | v<-vs ] (Str " ")

ppCases :: [([char],[[char]],expr [char])] → text
ppCases cases
    = ppList [ Block [Str c,Str " ",ppVars vs,
                      Str " -> ",Indent (pp 1 e)]
             | (c,vs,e)<-cases ] (Block [Str " |",NL])
```

This completes the presentation of the pretty-printer.

List of Tables

Let me see: four times five is twelve,
and four times six is thirteen, and four times seven is—oh dear!
I shall never get to twenty at that rate! However, the
Multiplication Table doesn't signify: let's try Geography.
London is the capital of Paris, and Paris is the capital of
Rome, and Rome—no, THAT'S all wrong, I'm certain!
— Lewis Carroll, *Alice's Adventures in Wonderland (1865) ch. 2*

List of Figures

'What is the use of a book', thought Alice, 'without pictures or conversations?'
— Lewis Carroll, *Alice's Adventures in Wonderland (1865) ch. 1*

Bibliography

Let us come now to references to authors. You have nothing else to do but look for a book that quotes them all, from A to Z. Then you put this same alphabet into yours. Perhaps there will even be someone silly enough to believe that you have made use of them all in your simple and straightforward story.

— Cervantes, *Don Quixote (1604)*

[1] *Proceedings of the 13th ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, New York, January 1986. ACM-Press.

[2] *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, New York, January 1987. ACM-Press.

[3] *FPCA '89: The Fourth International Conference on Functional Programming Languages and Computer Architecture, London, UK*, New York, September 1989. ACM-Press.

[4] *Proceedings of the 17th ACM Symposium on Principles of Programming Languages, San Francisco, California*, New York, January 1990. ACM-Press.

[5] *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, France*, New York, June 1990. ACM-Press.

[6] *Proceedings of the 18th ACM Symposium on Principles of Programming Languages, Orlando, Florida*, New York, January 1991. ACM-Press.

[7] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5–40, 1990.

[8] S. Abramsky and T.S.E. Maibaum, editors. *Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK*, number 494 in Lecture Notes in Computer Science, Berlin, April 1991. Springer Verlag.

[9] Samson Abramsky. Strictness analysis and polymorphic invariance (extended abstract). In Ganzinger and Jones [61], pages 1–23.

[10] Samson Abramsky and Chris Hankin. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Limited, Chichester, 1987.

[11] Samson Abramsky and Chris Hankin. An introduction to abstract interpretation. In *Abstract Interpretation of Declarative Languages* [10], chapter 1, pages 9–31.

[12] Samson Abramsky and Thomas P. Jensen. A relational approach to strictness analysis for higher-order polymorphic functions. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages, Orlando, Florida* [6], pages 49–54.

[13] Torben Amtoft. Strictness types: An inference algorithm and an application. Technical Report DAIMI PB-448, Computer Science Department, Aarhus University, Aarhus, DK, 1993.

[14] Torben Amtoft. Local type reconstruction by means of symbolic fixed point iteration. In Sannella [136], pages 43–57.

[15] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programms. *Communications of the ACM*, 21(8):613–641, August 1978.

[16] Gebreselassie Baraki. A note on abstract interpretation of polymorphic functions. In Hughes [84], pages 367–378.

[17] Nick Benton. *Strictness Analysis of Lazy Functional Programs*. PhD thesis, University of Cambridge Computer Laboratory, December 1992.

[18] P.N. Benton. Strictness logic and polymorphic invariance. In A. Nerode and M. Taitslin, editors, *The 2nd International Symposium on Logical Foundations of Computer Science — Tver '92*, number 620 in Lecture Notes in Computer Science, pages 33–44, Berlin, July 1992. Springer Verlag.

[19] P.N. Benton. Strictness properties of lazy algebraic datatypes. In Cousot et al. [42], pages 206–217.

[20] Gérard Berry. Stable models of typed λ-calculi. In *Proceedings of the 5th International Colloquium on Automata, Languages and Programming*, number 62 in Lecture Notes in Computer Science, pages 72–89, Berlin, 1978. Springer Verlag.

[21] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Series in Computer Science. Prentice Hall International, 1988.

[22] Adrienne Bloss. Path analysis and the optimization of nonstrict functional languages. *ACM Transactions on Programming Languages and Systems*, 16(3):328–369, 1994.

[23] Adrienne Bloss and Paul Hudak. Path semantics. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of the 3rd Workshop on Mathematical Foundations of Programming Language Semantics*, number 298 in Lecture Notes in Computer Science, New York, NY, April 1987. Springer Verlag.

[24] Adrienne Bloss and Paul Hudak. Variations on strictness analysis. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, France* [5], pages 132–142.

[25] Adrienne Bloss, Paul Hudak, and Jonathan Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1:147–164, 1988.

[26] Maurice Bruynooghe and Jaan Penjam, editors. *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming, Tallinn, Estonia*, number 714 in Lecture Notes in Computer Science, Berlin, August 1993. Springer Verlag.

[27] Randal E. Bryant. Symbolic Boolean manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[28] Geoffrey L. Burn. Evaluation transformers — a model for the parallel evaluation of functional languages (extended abstract). In Kahn [100], pages 446–470.

[29] Geoffrey L. Burn. A relationship between abstract interpretation and projection analysis (extended abstract). In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages, San Francisco, California* [4].

[30] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[31] G.L. Burn. Using projection analysis in compiling lazy functional programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, France* [5], pages 227–241.

[32] G.L. Burn. The evaluation transformer model of reduction and its correctness. In Abramsky and Maibaum [8], pages 458–482.

[33] Tyng-Ruey Chuang and Benjamin Goldberg. Backward analysis for higher-order functions using inverse images. Technical Report 620, Department of Computer Science, New York University, NY, February 1991.

[34] Tyng-Ruey Chuang and Benjamin Goldberg. A syntactic approach to fixed point computation on finite domains. *ACM LISP Pointers*, 5(1):109–118, September 1992.

[35] Chris Clack and Simon L. Peyton Jones. Strictness analysis — a practical approach. In Jouannaud [99], pages 35–49.

[36] Charles Consel. Fast strictness analysis via symbolic fixpoint iteration. In Le Charlier [110], pages 423–431.

[37] G. Cousineau, P.-L. Curien, and M. Mauny. The Categorical Abstract Machine. In Jouannaud [99], pages 50–64.

[38] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 237–277. North-Holland, Amsterdam New York Oxford, 1978.

[39] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Language, Los Angeles, California*, pages 238–252, New York, January 1977. ACM-Press.

[40] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 269–282, New York, 1979. ACM-Press.

[41] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming, Leuven, Belgium*, number 631 in Lecture Notes in Computer Science, pages 269–295, Berlin, 1992. Springer Verlag.

[42] Patrick Cousot, Moreno Falaschi, Gilberto Filè, and Antoine Rauzy, editors. *Third International Workshop On Static Analysis, Padova, Italy*, number 724 in Lecture Notes in Computer Science, Berlin, September 1993. Springer Verlag.

[43] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages, Albequerque, New Mexico*, pages 207–212, New York, January 1982. ACM-Press.

[44] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1992.

[45] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[46] Kei Davis. A note on the choice of domains for projection-based program analysis. In Heldal et al. [70], pages 73–81.

[47] Kei Davis. Analysing functions by projection-based backward abstraction. In John Launchbury and Patrick Sansom, editors, *Functional Programming, Glasgow 1992: Proceedings of the 1992 Workshop, Ayr, Scotland*, Workshops in Computing, pages 43–56, London, July 1993. Springer Verlag.

[48] Kei Davis. Projection-based termination analysis. In O'Donnell and Hammond [126], pages 26–42.

[49] Kei Davis and John Hughes, editors. *Functional Programming, Glasgow 1989 Proceedings of the 1989 Workshop, Fraserburgh, Scotland*, Workshops in Computing, London, UK, August 1990. Springer Verlag.

[50] Kei Davis and Philip Wadler. Backwards strictness analysis: Proved and improved. In Davis and Hughes [49], pages 12–30.

[51] Kei Davis and Philip Wadler. Strictness analysis in 4D. In Peyton Jones et al. [131].

[52] N.G. de Bruijn. A lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[53] Alain Deutsch. An operational model of strictness properties and its abstractions. In Heldal et al. [70], pages 82–99.

[54] P. Dybjer. Computing inverse images. In *Proceedings of the International Conference on Automata, Languages, and Programming, Karlsruhe, Germany*, number 267 in Lecture Notes in Computer Science, Berlin, 1987. Springer Verlag.

[55] Peter Dybjer. Inverse image analysis generalises strictness analysis. *Information and Computation*, 90(2):194–216, February 1991.

[56] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Kahn [100], pages 34–46.

[57] Alex Ferguson and John Hughes. Fast abstract interpretation using sequential algorithms. In Cousot et al. [42], pages 45–59.

[58] Sigbjørn Finne and Geoffrey Burn. Assessing the evaluation transformer model of reduction on the spineless G-machine. In *FPCA '93: The Sixth International Conference on Functional Programming Languages and Computer Architecture, Copenhagen, DK*, pages 331–169, New York, June 1993. ACM-Press.

[59] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In Harald Ganzinger, editor, *Proceedings of the 2nd European Symposium on Programming, Nancy, France*, number 300 in Lecture Notes in Computer Science, Berlin, March 1988. Springer Verlag.

[60] Jean H. Gallier. *Logic for Computer Science, Foundations of Automatic Theorem Proving*. Harper & Row, Publishers, Inc., 1986.

[61] Harald Ganzinger and Neil D. Jones, editors. *Proceedings of the Workshop on Programs as Data Objects, Copenhagen, Denmark*, number 217 in Lecture Notes in Computer Science, Berlin, October 1985. Springer Verlag.

[62] Carsten K. Gomard and Peter Sestoft. Evaluation order analysis for lazy data structures. In Heldal et al. [70], pages 112–127.

[63] Carsten K. Gomard and Peter Sestoft. Path analysis for lazy data structures. In Bruynooghe and Penjam [26], pages 54–68.

[64] C.A. Gunter and D.S. Scott. Semantic domains. In van Leeuwen [147], chapter 12, pages 633–674.

[65] Cordelia V. Hall and David S. Wise. Compiling strictness into streams. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany* [2], pages 132–143.

[66] Chris Hankin and Sebastian Hunt. Approximate fixed points in abstract interpretation. In Krieg-Brückner [104], pages 219–232.

[67] Chris Hankin and Daniel Le Métayer. Deriving algorithms from type inference systems: Application to strictness analysis. In *Proceedings of the 21th ACM Symposium on Principles of Programming Languages, Portland, Oregon*, pages 202–212, New York, January 1994. ACM-Press.

[68] Chris Hankin and Daniel Le Métayer. Lazy type inference for the strictness analysis of lists. In Sannella [136], pages 257–271.

[69] Robert Harper, Robin Milner, and Mads Tofte. The definition of standard ML, version 2. Technical Report ECS-LFCS-88-2, Department of Computer Science, University of Edinburgh, Edinburgh, UK, 1988.

[70] Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler, editors. *Functional Programming, Glasgow 1991: Proceedings of the 1991 Workshop, Portree, Isle of Skye*, Workshops in Computing, London, August 1992. Springer Verlag.

[71] Peter Henderson. *Functional Programming: Application and Implementation*. Prentice Hall International, 1980.

[72] Fritz Henglein. Iterative fixed point computation for type-based strictness analysis. In Le Charlier [110], pages 395–407.

[73] Ralf Hinze. *Einführung in die funktionale Programmierung mit Miranda*. Teubner, Stuttgart, 1992.

[74] Denis B. Howe and Geoffrey L. Burn. Using strictness in the stg machine. In O'Donnell and Hammond [126], pages 127–137.

[75] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5), 1992.

[76] Paul Hudak and Jonathan Young. Higher-order strictness analysis in untyped lambda calculus. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida* [1], pages 97–109.

[77] P.R. Hudak and J.H. Young. A set-theoretic characterization of function strictness in the lambda calculus. Research Report YALEU/DCS/RR-391, Yale University Department of Computer Science, New Haven, CT, 1985.

[78] John Hughes. Strictness detection in non-flat domains. In Ganzinger and Jones [61], pages 112–135.

[79] John Hughes. Analysing strictness by abstract interpretation of continuations. In *Abstract Interpretation of Declarative Languages* [10], chapter 4, pages 63–102.

[80] John Hughes. Abstract interpretations of first order polymorphic functions. In C.V. Hall, R.J.M. Hughes, and J.T. O'Donnell, editors, *Proceedings of the 1988 Glasgow Workshop on Functional Programming, Rothesay, Isle of Bute, Scotland*, pages 68–86, February 1989. Research Report 89/R4.

[81] John Hughes. Projections for polymorphic strictness analysis. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science, Manchester, UK, Proceedings*, number 389 in Lecture Notes in Computer Science, pages 82–100. Springer Verlag, Berlin, September 1989.

[82] John Hughes. Compile-time analysis of functional programs. In Turner [146], chapter 5, pages 117–154.

[83] John Hughes. Why functional programming matters. In Turner [146], chapter 2, pages 17–42.

[84] John Hughes, editor. *5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA*, number 523 in Lecture Notes in Computer Science, Harvard, Massachusetts, USA, August 1991. Springer Verlag.

[85] John Hughes and John Launchbury. Towards relating forwards and backwards analyses. In Peyton Jones et al. [131], pages 101–113.

[86] John Hughes and John Launchbury. Reversing abstract interpretations. In Krieg-Brückner [104], pages 269–286.

[87] R.J.M. Hughes. Backwards analysis of functional programs. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 187–208. North-Holland, Amsterdam, NL, 1988. Also Report CSC/87/R3, Department of Computing Science, University of Glasgow (1987).

[88] Sebastian Hunt. Frontiers and open sets in abstract interpretation. In *FPCA '89: The Fourth International Conference on Functional Programming Languages and Computer Architecture, London, UK* [3], pages 1–11.

[89] Sebastian Hunt. PERs generalise projections for strictness analysis. In Peyton Jones et al. [131], pages 114–125.

[90] Peter Z. Ingerman. Thunks — a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, January 1961.

[91] Kristian Damm Jensen, Peter Hjæresen, and Mads Rosendahl. Efficient strictness analysis of Haskell. In Le Charlier [110], pages 346–362.

[92] Thomas P. Jensen. Strictness analysis in logical form. In Hughes [84], pages 352–366.

[93] Thomas P. Jensen. Disjunctive strictness analysis. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 174–185, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[94] T. Johnsson. Detecting when call-by-value can be used instead of call-by-need. Technical Report Memo PMG-14, Programming Methodology Group, Institutionen för Informationsbehandling, Chalmers Tekniska Høgskola, Göteborg, SE, 1981.

[95] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jouannaud [99], pages 190–203.

[96] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs: Abridged version. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida* [1], pages 296–306.

[97] Simon B. Jones and Daniel Le Métayer. A new method for strictness analysis on non-flat domains. In Davis and Hughes [49], pages 1–11.

[98] Niels Jørgensen. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In Le Charlier [110], pages 327–345.

[99] Jean-Pierre Jouannaud, editor. *Functional Programming Languages and Computer Architecture, Nancy, France*, number 201 in Lecture Notes in Computer Science. Springer Verlag, September 1985.

[100] Gilles Kahn, editor. *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA*, number 274 in Lecture Notes in Computer Science, Berlin, September 1987. Springer Verlag.

[101] Samuel Kamin. Head-strictness is not a monotonic abstract property. *Information Processing Letters*, 41:195–198, 1992.

[102] Michael Kettler. Generische Striktheitsanalyse. Master's thesis, University of Bonn, April 1995.

[103] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is Dexptime-complete. In *Proceedings of the 15th Colloquium on Trees in Algebra and Programming*, number 431 in Lecture Notes in Computer Science, pages 206–220. Springer Verlag, 1990.

[104] B. Krieg-Brückner, editor. *Proceedings of the 4th European Symposium on Programming, Rennes, France*, number 582 in Lecture Notes in Computer Science, Berlin, February 1992. Springer Verlag.

[105] Ryszard Kubiak, John Hughes, and John Launchbury. Implementing projection-based strictness analysis. In Heldal et al. [70].

[106] Tsung-Min Kuo and Prateek Mishra. On strictness and its analysis. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany* [2], pages 144–155.

[107] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *FPCA '89: The Fourth International Conference on Functional Programming Languages and Computer Architecture, London, UK* [3], pages 260–272.

[108] Kirsten Lackner Solberg. Strictness and totality analysis with conjunction. In Peter D. Mosses, Mogens Nielson, and Michael I. Schwartzbach, editors, *Proceedings of the 6th International Joint Conference on Theory and Practice of Software Development, Aarhus, Denmark*, number 915 in Lecture Notes in Computer Science, pages 501–515, Berlin, May 1995. Springer Verlag.

[109] Kirsten Lackner Solberg, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. In Le Charlier [110], pages 408–422.

[110] Baudoin Le Charlier, editor. *Proceedings of the First International Static Analysis Symposium, Namur, Belgium*, number 864 in Lecture Notes in Computer Science, Berlin, September 1994. Springer Verlag.

[111] D.J. Lillie and P.G. Harrison. A projection model of types. In Hughes [84], pages 259–288.

[112] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Computation*, 71(1/2):95–130, 1986.

[113] Chris Martin and Chris Hankin. Finding fixed points in finite lattices. In Kahn [100], pages 426–445.

[114] Laurent Mauborgne. Abstract interpretation using TDGs. In Le Charlier [110], pages 363–379.

[115] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[116] P.D. Mosses. Denotational semantics. In van Leeuwen [147], chapter 11, pages 575–631.

[117] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In G. Goos and J. Hartmanis, editors, *Proceedings of the International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[118] Alan Mycroft. Completeness and predicate-based abstract interpretation. In Schmidt [137].

[119] Alan Mycroft and Neil D. Jones. A relational framework for abstract interpretation. In Ganzinger and Jones [61], pages 156–171.

[120] F. Nielson. Tensor products generalize the relational data flow analysis method. In M. Arató, I. Kátai, and L. Varga, editors, *Proceedings of the 4th Hungarian Computer Science Conference, Gyor, Hungary*, pages 211–225, Budapest, July 1985. Académiai Kiadó.

[121] Flemming Nielson. Strictness analysis and denotational abstract interpretation (extended abstract). In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany* [2], pages 120–131.

[122] Flemming Nielson. Strictness analysis and denotational abstract interpretation. *Information and Computation*, 76(1):29–92, January 1988.

[123] Flemming Nielson and Hanne Riis Nielson. Finiteness conditions for fixed point iteration. *ACM LISP Pointers*, 5(1):96–108, 1992.

[124] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[125] Hanne Riis Nielson and Flemming Nielson. Context information for lazy code generation. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, France* [5], pages 251–263.

[126] John T. O'Donnell and Kevin Hammond, editors. *Functional Programming, Glasgow 1993: Proceedings of the 1993 Workshop, Ayr, Scotland*, Workshops in Computing, London, July 1994. Springer Verlag.

[127] Richard A. O'Keefe. A smooth applicative merge sort. Research paper 182, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK, 1982.

[128] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[129] Simon Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. In O'Donnell and Hammond [126], pages 201–221.

[130] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice Hall International, 1987.

[131] Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors. *Functional Programming, Glasgow 1990 Proceedings of the 1990 Workshop, Ullapool, Scotland*, Workshops in Computing, London, UK, August 1991. Springer Verlag.

[132] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages: A Tutorial*. Series in Computer Science. Prentice Hall International, 1992.

[133] Gordon Plotkin. Domains. TEXed edition of the course note "Domains" by Gordon Plokin, prepared by Yugo Kashiwagi and Hidetaka Kondoh, 1992.

[134] Uday S. Reddy and Samuel N. Kamin. On the power of abstract interpretation. *Computer Languages*, 19(2):79–89, 1993.

[135] Mads Rosendahl. Higher-order chaotic iteration sequences. In Bruynooghe and Penjam [26], pages 332–345.

[136] Donald Sannella, editor. *Programming Languages and Systems — ESOP'94, Edinburgh, U.K.*, number 788 in Lecture Notes in Computer Science, Berlin, April 1994. Springer Verlag.

[137] D. Schmidt, editor. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, Copenhagen, Denmark*, New York, June 1993. ACM-Press.

[138] R.C. Sekar, Prateek Mishra, and I.V. Ramakrishnan. On the power and limitation of strictness analysis based on abstract interpretation. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages, Orlando, Florida* [6], pages 37–48.

[139] R.C. Sekar, Shaunak Pawagi, and I.V. Ramakrishnan. Small domains spell fast strictness analysis. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages, San Francisco, California* [4], pages 169–183.

[140] Peter Sestoft. *Analysis and efficient implementation of functional programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[141] Julian Seward. Polymorphic strictness analysis using frontiers. In Schmidt [137], pages 186–193.

[142] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[143] R.D. Tennent. *Semantics of Programming Languages*. Prentice-Hall, Englewood Cliffs, 1991.

[144] Peter Thiemann. *Grundlagen der funktionalen Programmierung*. Teubner, Stuttgart, 1994.

[145] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In Jouannaud [99], pages 1–16.

[146] David A. Turner, editor. *Research Topics in Functional Programming*. Addison-Wesley Publ. Comp., Inc., Reading, Massachusetts, 1990.

[147] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier Science Publishers B.V. (North Holland), 1990.

[148] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In *Abstract Interpretation of Declarative Languages* [10], chapter 12, pages 266–275.

[149] Philip Wadler. *Efficient Compilation of Pattern-Matching*, chapter 5, pages 78–103. In *Series in Computer Science* [130], 1987.

[150] Philip Wadler and R.J.M. Hughes. Projections for strictness analysis. In Kahn [100], pages 385–407.

[151] Pierre Weis, María-Virginia Aponte, Alain Laville, Michel Mauny, and Ascánder Suárez. The CAML reference manual, Version 2.6. Technical report, Projet Formel, INRIA-ENS, 1989.

[152] S.C. Wray. A new strictness detection algorithm. In L. Augustsson et al., editors, *Aspens Workshop on Implementation of Functional Languages, Göteborg*, Programming Methodology Group Report 17, University of Göteborg and Chalmers University of Technology, 1985, 1985.

[153] David A. Wright. A new technique for strictness analysis. In Abramsky and Maibaum [8].

Notation Index

The symbol := denotes ‘equals by definition’. Accordingly :$\Longleftrightarrow$ denotes ‘equivalent by definition’. The end of a proof is marked by ■.

Index

Page numbers given in boldface refer to definitions.

Curriculum Vitae

Ralf Thomas Walter Hinze
Eudenbergerstraße 13
53639 Königswinter

geboren am 2. Juli 1965 in Marl,
Familienstand: ledig, ein Kind.

1971 – 1975	August-Döhr-Grundschule in Marl
1975 – 1984	Albert-Schweitzer-Gymnasium in Marl
Apr. 1984	Abitur
1984 – 1990	Studium der Informatik mit Nebenfach theoretische Medizin an der Universität Dortmund
Mai 1989	Einreichung der Diplomarbeit „*Typsysteme und Typinferenzsysteme*“
Apr. 1990	Abschluß des Studiums
Apr. 1990 – Sep. 1990	Wissenschaftlicher Angestellter an der Abteilung Informatik der Universität Dortmund
seit Okt. 1990	Wissenschaftlicher Angestellter an der Abteilung Informatik der Universität Bonn