Abstract Interpretation and Attribute Grammars

Mads Rosendahl

Clare Hall
University of Cambridge

1991
A dissertation submitted for the degree of
Doctor of Philosophy

Summary

The objective of this thesis is to explore the connections between abstract interpretation and attribute grammars as frameworks in program analysis.

Abstract interpretation is a semantics-based program analysis method. A large class of data flow analysis problems can be expressed as non-standard semantics where the "meaning" contains information about the runtime behaviour of programs. In an abstract interpretation the analysis is proved correct by relating it to the usual semantics for the language.

Attribute grammars provide a method and notation to specify code generation and program analysis directly from the syntax of the programming language. They are especially used for describing compilation of programming languages and very efficient evaluators have been developed for subclasses of attribute grammars.

By relating abstract interpretation and attribute grammars we obtain a closer connection between the specification and implementation of abstract interpretations which at the same time facilitates the correctness proofs of interpretations.

Implementation and specification of abstract interpretations using circular attribute grammars is realised with an evaluator system for a class of domain theoretic attribute grammars. In this system thecircularity of attribute grammars is resolved by fixpoint iteration. The use of finite lattices in abstract interpretations requires automatic generation of specialised fixpoint iterators. This is done using a technique called lazy fixpoint iteration which is presented in the thesis.

Methods from abstract interpretation can also be used in correctness proofs of attribute grammars. This proof technique introduces a new class of attribute grammars based on domain theory. This method is illustrated with examples.

Preface

I would like to thank my supervisor Alan Mycroft for all the help. It was his idea to look into the connection between attribute grammars and abstract interpretation. I am especially grateful for his assistance during the final stage of the work.

I would also like to thank Troy Ferguson for careful proof readings of drafts of the thesis.

I am grateful for the assistance given by the members of staff of the Computer Laboratory, especially Twiggy and Edi for extra coffee, biscuits, and insults. I would like to thank Juanito Camilleri, Roy Crole, and Andy Gordon for their friendship during my stay in Cambridge.

The thesis would not have been the same without the intellectual environment provided by Clare Boat Club.

My time in Cambridge enriched me with at least one major discovery: Troy. Without her, life (and thesis work) would not have been as exciting.

The following bodies provided financial assistance for subsistence and travel for which I am grateful: University of Copenhagen, the Natural Science Research Council of Denmark, and Forskerakademiet (Denmark).

The main part of chapter 6 has been published as:

Mads Rosendahl
Abstract Interpretation using Attribute Grammars.
In *Workshop on Attribute Grammars and their Applications,Paris, France*
(P Deransart and M Jourdan, eds.), pp. 143–156.
Volume 461 of Lecture Notes in Computer Science.
Springer-Verlag, Oct., 1990.

Parts of chapter 3 has been published as:

Mads Rosendahl
Automatic Complexity Analysis.
In *Functional Programming and Computer Architecture, London, England*, pp. 144–156.
ACM Press, Sept., 1989.

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

I hereby declare that this dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

I further state that no part of my dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

Mads Rosendahl

Contents

1 Introduction **1**
1.1 Concepts and notation . . . 2
1.2 Purpose of the thesis . . . 3
1.3 History . . . 5
1.3.1 Data flow analysis . . . 5
1.3.2 Abstract interpretation . . . 5
1.3.3 Attribute grammars . . . 8
1.4 Outline . . . 10

2 Domains and Grammars **13**
2.1 Domain theory . . . 13
2.1.1 Complete partially ordered sets . . . 13
2.1.2 Fixpoint theorem . . . 16
2.1.3 Domain constructors . . . 16
2.1.4 Recursive domain equations . . . 20
2.1.5 Lattices . . . 23
2.2 Inclusive relations . . . 23
2.2.1 Logical relations . . . 24
2.2.2 Inclusive relations on recursive domains . . . 26
2.2.3 Infinite lists . . . 27
2.3 Grammars . . . 29
2.3.1 Context-Free Grammar . . . 30
2.3.2 Algebraic description . . . 31
2.3.3 Abstract syntax . . . 32

3 Frameworks for Abstract Interpretation **35**
3.1 Fixpoint semantics . . . 35
3.1.1 Collecting semantics . . . 37
3.1.2 Meet over all paths . . . 38
3.1.3 Independent attribute method . . . 38
3.1.4 Relational method . . . 40
3.1.5 Minimal function graph . . . 41
3.1.6 Instrumented semantics . . . 43

3.1.7 Powerdomains . . . 45
3.2 Abstraction/Concretisation . . . 46
3.2.1 Embedding . . . 46
3.2.2 Abstract domain . . . 46
3.2.3 Correctness . . . 47
3.2.4 Fixpoint induction . . . 47
3.2.5 Induced analysis . . . 48
3.2.6 Constant propagation . . . 49
3.2.7 Inclusive Relations . . . 51
3.3 Relational abstract interpretation . . . 53
3.3.1 Language . . . 53
3.3.2 Type assignment . . . 54
3.3.3 Domain assignment . . . 55
3.3.4 Meaning assignment . . . 56
3.3.5 Versions and interpretations . . . 57
3.3.6 Relation Assignment . . . 57
3.3.7 Abstraction Theorem . . . 58
3.3.8 A model for polymorphism . . . 58
3.4 Automatic complexity analysis . . . 58

4 A Metalanguage for Abstract Interpretation 62
4.1 Computability . . . 63
4.1.1 Lambda-definability . . . 63
4.1.2 Decidability . . . 63
4.1.3 Effective domains . . . 64
4.1.4 Fixpoints . . . 64
4.1.5 The bottom element . . . 66
4.1.6 Metalanguage . . . 67
4.2 A simple typesystem . . . 67
4.2.1 Implementation language . . . 68
4.2.2 Fixpoints . . . 69
4.2.3 Lifted domains . . . 69
4.3 Finite domains . . . 70
4.3.1 Composite domains . . . 71
4.4 Lazy fixpoint iteration . . . 72
4.4.1 Memo-function . . . 72
4.4.2 Function domain . . . 74
4.4.3 Fixpoint iteration for other domains . . . 78
4.5 A domain-theoretic language . . . 79
4.5.1 The simple type structure . . . 79
4.5.2 Finite height domains . . . 79
4.5.3 Combination . . . 80
4.5.4 Metalanguage . . . 81

4.5.5 Implementation . . . 82
4.5.6 Conclusion . . . 83

5 Attribute Grammars as Semantic Descriptions 85
5.1 Attribute Grammars . . . 85
5.1.1 Notation . . . 86
5.1.2 Extended attribute grammars . . . 86
5.1.3 Attributed production . . . 87
5.1.4 Semantics . . . 88
5.1.5 Evaluation . . . 90
5.2 Background . . . 90
5.2.1 Semantics of attribute grammars . . . 91
5.2.2 Attribute grammars as semantics . . . 92
5.2.3 Attribute evaluation . . . 92
5.2.4 Compilers . . . 93
5.3 Logical attribute grammars . . . 94

6 Correctness Proofs of Attribute Grammars 95
6.1 Domain attribute grammars . . . 95
6.1.1 Domain attribute grammars . . . 95
6.1.2 Attributed scheme . . . 96
6.1.3 Fixpoint induction . . . 97
6.2 A small language . . . 97
6.3 Standard interpretation . . . 98
6.3.1 Attributed scheme . . . 98
6.3.2 Standard interpretation . . . 100
6.4 Live-variable analysis . . . 101
6.5 Correctness proof of the abstract interpretation . . . 102
6.5.1 Relation . . . 102
6.5.2 Local correctness . . . 102
6.5.3 Fixpoint induction . . . 103
6.6 Implementation of the attribute grammar . . . 104

7 Implementation of Abstract Interpretation using Attribute Grammars 105
7.1 Description . . . 105
7.2 Specification . . . 106
7.3 Proof rules . . . 108
7.4 Implementation . . . 108
7.5 Comparison . . . 109
7.6 Example . . . 110

8 Conclusion 115

References **117**

Index **129**

Last page **132**

Chapter 1

Introduction

Abstract interpretation is a semantics-based program analysis method. The semantics of a programming language can be specified as a mapping of programs to mathematical objects that describes the input-output function for the program. In an abstract interpretation the program is given a mathematical meaning in the same way as with a normal semantics. This however is not necessarily the standard meaning but it can be used to extract information about the computational behaviour of the program.

In this thesis we will explore the connections between abstract interpretation and attribute grammars as frameworks in program analysis. Attribute grammars provide a method and notation to specify code generation and program analysis directly from the syntax of the programming language.

Abstract interpretation The central idea in abstract interpretation is to construct two different meanings (semantics or interpretations) of a programming language where the first gives the usual meaning of programs in the language, and the other can be used to answer certain questions about the runtime behaviour of programs in the language.

The standard meaning of programs can typically be described by their input-output function and the standard interpretation will then be a function $\mathbf{I}_1$ which maps programs to their input-output functions.

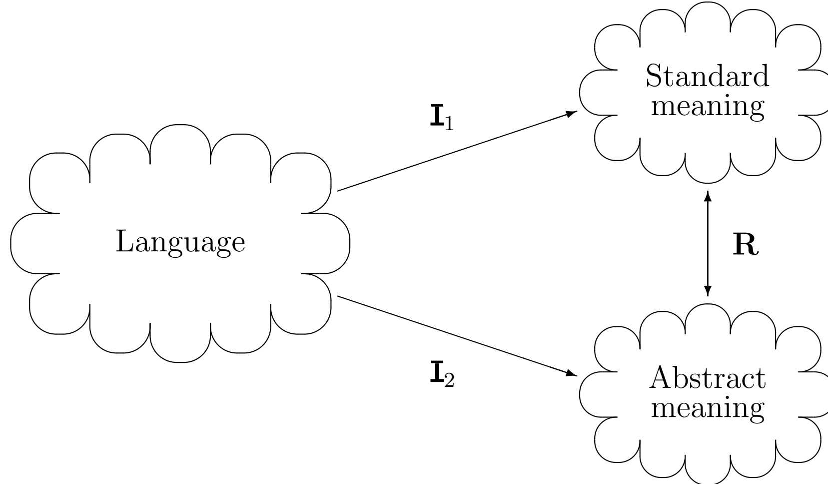
The abstract meaning will be defined by a function $\mathbf{I}_2$ which maps programs to mathematical objects that can be used to answer the question raised by a program analysis problem.

The correctness (or soundness) of this approach to program analysis can be established by proving a relation between these two interpretations. An abstract interpretation of a language consists of the two function $\mathbf{I}_1$ and $\mathbf{I}_2$ together with a relationship $\mathbf{R}$ between the meanings provided by these functions such that for all programs $\mathbf{p}$ the relation holds:

$$\mathbf{I}_1[\![\mathbf{p}]\!]\ \mathbf{R}\ \mathbf{I}_2[\![\mathbf{p}]\!]$$

The relationship **R** between the two meanings describes which real program behaviours are described by an abstract meaning.

This can be sketched as



Altogether an abstract interpretation consists of four different parts: the two interpretations ($\mathbf{I}_1$ and $\mathbf{I}_2$), the relation (**R**), and finally some use of the abstract meaning. The last part is normally the least formal. The abstract information is often used to guide a transformation or implementation of the program being analysed, and it is then argued that the resulting program behaves equivalently to the original.

The aim in abstract interpretation is twofold. We prove the correctness of the abstract semantics with respect to the standard semantics and then use the specification of the semantics as a basis for an implementation. Compared to more ad hoc methods of program analysis such a framework may add a high degree of reliability and a structure which facilitates the specification of more complex analyses.

This general scheme has been used in a number of variations. The two meanings of a program are normally expressed in the form of fixed point semantics and the relationship can be constructed from functions which map abstract meanings to sets of possible standard meanings.

1.1 Concepts and notation

A number of words commonly used in this area have several meanings. This overloading of meanings is a common cause of confusion in the literature.

We may talk about *attribute grammars* as a formalism for describing the semantics of context-free grammars and about a specific *attribute grammar* as a grammar with associated *attribute definitions*.

The phrase *abstract interpretation* is frequently used both as the name of the method and as the actual interpretation of a program in terms of abstract meaning. We will try to use it only about the method and use *abstract semantics* to refer to the interpretation as abstract meanings.

Both attribute grammars and abstract interpretation are *formalisms*—that is notations or languages for giving meaning or describing translations. At the same time they are the basis for implementations in which case they are *solution techniques*.

Similar confusion exists about the phrase *program analysis*. Abstract interpretation is often used to solve *program analysis problems*. This means that an implementation of an abstract interpretation is used to extract *data flow information* about programs. We may say that we perform a *program analysis* or that the program is *analysed*. This information may afterwards be used to compile the program or to direct a *program transformation*.

A number of other concepts in the thesis will be used with a somewhat specialised meaning. We list a few of the more important definitions here but a further discussion can be found later in the thesis.

Category is only used in the sense of *syntactic category*. That is the set of tokens (or lexical units) in the language.

Domain means *complete partially ordered set* (or *cpo* for short).

Standard semantics means the usual or "standard" meaning normally associated with a program. In denotational semantics *standard semantics* is often used with the connotation of *continuation semantics*. We will, however, use the phrase to distinguish it from the non-standard or abstract semantics provided as part of an abstract interpretation.

The phrase *collecting semantics* has a number of different meanings. This is discussed further in section 3.1.

Notation In the thesis we will use upper case *italics* letters to denote arbitrary sets and lower case *italics* letters to denote elements in sets. We will use the usual mathematical sloppiness of saying "the set **S**" to mean the set denoted by (the symbol) **S**. It is only computer science that has taught us the importance of distinguishing between a variable and the value of a variable (plus the location of the variable and the representation of the value of the variable).

1.2 Purpose of the thesis

The main objective of this thesis is to investigate the connection between abstract interpretation and attribute grammars. Both methods have been used for program analysis but, where most work in abstract interpretation has dealt with the correctness proofs of various analyses, the main emphasis in attribute grammars has been to identify classes of grammars with space- and time-efficient implementations.

Attribute grammars share some structural similarities with abstract interpretation in the way information is synthesised from the program text. It is furthermore a technique which over the last decades has gained considerable success as a compiler writing method for real programming languages. This raises two natural secondary applications: to use the results from attribute grammars to simplify implementation of abstract interpretation and also to use abstract interpretation to prove attribute grammars correct.

The incentive for this thesis comes from experiences in the use of abstract interpretation. The typical application of abstract interpretation is based on denotational semantics but in addition it is necessary to set up a framework and introduce a notion of soundness (correctness) specific for the application. This can be acceptable when one wants to prove that a certain data-flow analysis can be proved correct but there is a feeling of repeated work when abstract interpretation is used as a way to achieve reliable program analysis tools. Moreover, when the interpretation has been specified, there is still the implementation to consider.

Eventhough abstract interpretation is a powerful tool in program analysis its use outside experimental systems is still rather limited. This may because of the the technical complications in the implementation and the mathematical requirements for setting up a sound framework for proving correctness.

One of the central aims of this thesis is to develop rules and methods for the specification, proof, and implementation of abstract interpretation which can be used for a variety of languages and program analysis problems. In this way we hope to make it easier to solve data-flow analysis problems in the framework of abstract interpretation without relaxing the correctness criterion.

There are typically two different ways to use attribute grammars in compiler construction. They may be used as a specification language which is then, by hand, transformed into a syntax-directed translation scheme that can be implemented using an automatically generated parser. The second possibility is to use an attribute grammar system such that the attribute grammar can be implemented directly.

The second alternative is often used for prototyping as it is occasionally argued that these systems cannot compete with handcoded compilers. It is not always clear whether there is any substance to this criticism but their use in prototyping alone make attribute grammars interesting.

Abstract interpretation is normally only used as a specification and proof technique. The implementation of an abstract semantics as a data flow analysis is a separate phase typically done by hand. It may be possible directly to convert the semantics into a functional program but in other situations specialised techniques are needed in the implementation.

One of the purposes of this thesis is to explore the possibility of using techniques from attribute grammars to automate the implementation of abstract interpretation. A system based on this could be used as a prototyping tool and as a system to explore more efficient analysis techniques.

Combining the attribute grammar framework with abstract interpretation introduces a new way to prove the correctness of an attribute grammar. The standard semantics will be defined in an extended class of attribute grammars, powerful enough to describe the kinds of objects used in such a semantics. The abstract interpretation may belong to one of the more restrictive class of attribute grammars which can be implemented efficiently.

1.3 History

In this section we will give an overview of the main literature which has influenced the work reported in this thesis. This will not be a complete survey of the literature in this area but rather an outline of the background for the work.

1.3.1 Data flow analysis

The field of abstract interpretation has its origin in optimising compilers under the name of *data flow analysis*.

Possibly the first work to use the ideas now known as abstract interpretation was [Naur 1965] who made a so-called pseudo evaluation of Algol. The analysis extracted type information from expressions as a formalisation of typechecking in Algol programs. The method was used to identify implicit type conversion in expressions.

Data flow analysis was the immediate predecessor to abstract interpretation. [Hecht 1977] contains an overview of the approach and the state of the technology from shortly before the more semantically based abstract interpretation method was introduced. Data flow analysis uses the control flow structure of the program to collect information about values and use of variables in programs. There is a distinction between *interprocedural* and *intraprocedural* data flow analysis. The first phrase covers analysis of the program seen as a single entity. The second and simpler (but also less powerful) approach is to analyse blocks or procedures one at a time without propagating information from one block to the next. Some of the works from this period which have influenced abstract interpretation are [Rosen 1980], [Wegbreit 1975], [Fosdick & Osterweil 1976], and [Kildall 1973]. A bibliography on flow analysis can be found in [Muchnick & Jones 1981] and [Marlowe & Ryder 1990] gives an overview of the more recent results in this area.

1.3.2 Abstract interpretation

The formalisation of a proof technique for data flow analysis was given by [Cousot & Cousot 1977] under the name of *abstract interpretation*. The framework was further extended in [Cousot & Cousot 1979] and [Cousot 1981] and applications within a wide range of areas were described.

The first work on abstract interpretation of applicative programming languages was [Mycroft 1981]. The generalisation to abstract interpretation of denotational language definitions was made by [Nielson 1984], and in its expressive power this encompasses the former works, at least for forward analysis.

New types of analysis have proved to be useful especially in compilers for functional programming languages. Strictness analysis and type-checking [Damas & Milner 1982] are now normal parts of compilers for polymorphically typed, lazy languages and the formal basis is given as a relation to a standard semantics. Optimisations of generated code will, in these examples, be program transformations that use the result of a program analysis. The analysis itself can be expressed as an abstract interpretation, *i.e.* by interpreting the source code in two different ways: the standard meaning and another meaning which can be used in the program transformation.

Strictness analysis One of the most important analysis techniques developed within the framework of abstract interpretation is *strictness analysis* [Mycroft 1980]. The aim is to find when call-by-need in lazy functional programming languages can be replaced by call-by-value. Over the last decade the technique has been extended to higher-order functions [Burn, Hankin & Abramsky 1986], [Hudak & Young 1988], structured data ([Wadler 1987], [Hughes 1988], [Wadler & Hughes 1987]), and polymorphic types [Abramsky 1986]. As an extension of strictness analysis [Bloss & Hudak 1986] have considered other types of questions about the order of evaluation in a lazy functional language.

Applications of abstract interpretation Strictness analysis is only one example of an abstract interpretation. Many other program analysis problems have been solved in this framework. For functional programming languages the examples include destructive updates of arrays ([Hudak & Bloss 1985], [Bloss & Hudak 1987]), complexity analysis [Rosendahl 1989], and compile time garbage collection ([Hudak 1987],[Jensen & Mogensen 1990]).

Denotational abstract interpretation A more general approach to abstract interpretation is to base it directly on semantic descriptions. This method has been pursued by Flemming Nielson in a number of articles (*eg.* [Nielson 1984], [Nielson 1986a], [Nielson 1989]). The general aim is make non-standard interpretations of the metalanguage which is used in denotational semantics. An ordinary denotational semantics can then be given two different interpretations where one is the standard meaning and the other is an abstract meaning. In this framework the interpretation is language independent and all languages with a denotational semantics can be analysed using the same interpretation. Unfortunately it is difficult to specify interpretations which are powerful enough to be used for all languages and it is

not always clear whether an interpretation specified this way is an analysis of the language or of the metalanguage.

Frameworks for abstract interpretation As it is difficult to find a framework for abstract interpretation which can be used for all languages some more specialised techniques have been developed for certain languages.

For lazy higher-order functional languages [Jones 1987a] has developed a method based on tree grammars to perform data flow analysis. For lazy first-order languages with structured data [Wadler & Hughes 1987] have developed a method based on *projections* for strictness analysis.

Analysis of logical programming languages has developed into a separate field which has little in common with the analysis of functional programming languages. An analysis of Prolog can either be semantics-based with a bottom-up technique [Marriott & Søndergaard 1989], or it may use an operational top-down method [Bruynooghe et al 1987].

The analysis of imperative languages can also be based on operational semantics as exemplified in [Bourdoncle 1990] and [Deutsch 1990] for alias analysis of reference parameters.

Implementation Abstract interpretation has been used in a number of compiler generators. The PSI-project ([Nielson & Nielson 1988], [Nielson 1987]) is based on the implementation of a metalanguage in a special semantics where the "meaning" is the code to be generated. The MIX project ([Jones, Sestoft & Søndergaard 1985], [Jones, Sestoft & Søndergaard 1989]) uses abstract interpretation for partial evaluation in the phase called *binding time analysis* [Bondorf et al 1988].

Efficient implementation of abstract interpretation has not attracted much interest. Only recently has work on finding fixpoints in finite lattices been published. For strictness analysis the abstract meaning can be represented more efficiently using so called frontiers (see [Martin & Hankin 1987], [Jones & Clack 1987]), [Hunt 1989], and [Jones 1987b]). Other techniques include pending analysis [Young & Hudak 1986] and widening [Cousot & Cousot 1977].

Relations In the original definition of abstract interpretation [Cousot & Cousot 1977] the proof technique used *abstraction/concretisation* functions. This may be too restrictive for some applications. Techniques based on inclusive relations ([Statman 1980], [Statman 1985], [Manna, Ness & Vuillemin 1972]) may simplify proofs and their use in abstract interpretation has been explored by [Abramsky 1990], [Mycroft & Jones 1986]), and [Nielson 1984].

Correctness proofs of abstract interpretations may be seen as a special case of more general equivalence proofs of semantics. An example of this type of proof is the equivalence between an operational and a denotational semantics ([Reynolds 1974] and [Stoy 1977]) or between static and dynamic semantics (*eg.* [Despeyroux

1986]) and to describe polymorphism ([Reynolds 1983]). Such proofs may be based on inclusive relations and further results of their existence has been explored by [Mulmuley 1985] in the context of full abstraction.

The basic idea is to use relations between two interpretations instead of abstraction/concretisation functions. The relations are defined for the ground types used in the interpretations and are then extended to higher-order types according to a logical scheme. Milne and Reynolds use such relations in the equivalence proofs of semantics definitions. Plotkin uses logical relations to characterise λ-definability by classes of the typed λ-calculus. [Reynolds 1983] uses a relational approach to describe polymorphism and data abstraction. The latter paper uses a framework which is similar in style to the specification style of abstract interpretation.

There is no clear distinction between these examples and what is found in abstract interpretation. Typically, however, the abstract meaning can be realised effectively in the sense that it is finite and evaluation of it is guaranteed to terminate. This is often expressed as a requirement to the domain of abstract meanings to be finite or have finite height.

Foundation The mathematical foundation of abstract interpretation is in denotational semantics ([Gordon 1979], [Schmidt 1986], [Stoy 1977]) and domain theory ([Plotkin 1982], [Wadsworth 1978], [Smyth & Plotkin 1982]).

Abstract interpretation There is now an extensive literature covering applications in program optimisation, strictness analysis, binding time analysis, partial evaluation, code generation, complexity analysis, and so on. [Abramsky & Hankin 1987] contains a comprehensive bibliography of the area.

1.3.3 Attribute grammars

Attribute grammars have been widely investigated for efficient implementations and a large number of attribute evaluators has been developed for subsets of attribute grammars.

Attribute grammars were introduced by [Knuth 1968] (see also [Knuth 1971]) as an extension to syntax-directed compilation [Irons 1961]. A survey on results and implementations of attribute grammars can be found in [Deransart, Jourdan & Lorho 1988].

Attribute grammars and semantics The semantics of attribute grammars has been examined by [Chirica & Martin 1979] and [Mayoh 1981]. This shows a connection between attribute grammars and denotational semantics and further results on transforming denotational semantics into attribute grammars were reported in [Ganzinger 1980].

The relationship between logic programming and attribute grammars is discussed in [Deransart & Maluszynski 1985] and techniques to prove attribute grammars correct with respect to a specification is described in [Deransart 1983] and [Courcelle & Deransart 1988].

Extensions to attribute grammar notation Several different formalisms for attribute grammars have been proposed. Attribute coupled grammars ([Ganzinger & Giegerich 1984] and [Giegerich 1988]) use a uniform description of syntax (source programs) and semantics (attribute values). This means that such grammars can be composed. Further, a compiler may be specified in a modular fashion as a sequence of attribute coupled grammars. Higher-order attribute grammars extends this idea in that [Vogt, Swierstra & Kuiper 1989] attribute values themselves may be structure trees which may be evaluated by its own attribute grammar. Higher-order abstract syntax [Pfenning & Elliott 1988] is a generalisation of abstract syntax which can be used to propagate context-sensitive information in a style similar to attribute grammars.

Extended attribute grammars ([Watt & Madsen 1983] and [Madsen 1980]) is a more readable notation for attribute grammars which makes it more attractive as a semantic description language.

Attribute grammars with circular attribute dependencies arise naturally in data flow analysis [Sagiv et al 1989], VLSI design [Jones & Simon 1986], and semantics [Paulson 1982]. The circularity may either be removed by constructing a new attribute grammar with more attributes or attributes may be found by fixpoint iteration [Farrow 1986].

Attribute evaluators There are three main techniques for implementing attribute grammars. Some attribute grammars can be evaluated directly during parsing. This is the most efficient technique but it is only applicable for a small class of attribute grammars (L-attribute grammars: see [Deransart, Jourdan & Lorho 1988] and [Aho, Sethi & Ullman 1986]). Attribute grammars which cannot be evaluated during parsing require a second phase of evaluation. The most efficient evaluators of this kind determine an evaluation order for attributes directly from the attribute grammar, independently of the source program. This is done using so-called *visit-sequences* for attributes and the method is applicable to the class of *ordered attribute grammars*. A number of classes of attribute grammars where this can be achieved is described in [Deransart, Jourdan & Lorho 1988]. More general methods for attribute evaluation is often based on constructionof parse trees with slots for attributes. The attributes are then evaluated by repeated tree traversals.

The third and most general class of attribute grammars does not restrict dependencies of attributes as long as they are not circular. This means that the evaluation order must be determined on runtime dependent on the source program. An evaluation technique for this class consists of transforming the attribute grammar into

a recursion equation system (see [Jourdan 1984], [Engelfriet 1984], [Gallier, Manion & McEnerney 1985], [Gallier 1984], and [Filé 1986]). The recursive evaluation of attribute grammars is more easily performed in a lazy functional language as described in [Johnsson 1987].

There is a large number of evaluation systems for various classes of attribute grammars. The Synthesizer Generator [Reps & Teitelbaum 1984] uses incremental attribute evaluation and the system can be used to generate syntax-directed editors.

The Ergo attribute system ([Lee et al 1988] and [Nord & Pfenning 1989]) uses attribute grammars for semantic analysis of programs in a program derivation system. For efficiency this requires incremental attribute evaluation and sharing of attribute values since many versions of a program or specification may be present at the same time.

Analysis Program analysis in the framework of attribute grammars has been examined by [Babich & Jazayeri 1978] and [Wilhelm 1981]. This, however, requires some extensions to the usual notion of attribute grammars so as to realise iteration of information.

Compiler generation Attribute grammars can be seen as a generalisation of the syntax-directed translation that can be expressed using an attribute stack in some parsers [Johnson 1975]. Synthesised attribute evaluation in a functional language ([Uddeborg 1988] and [Jouvelot 1986]) extends the power so that inherited attributes can be simulated.

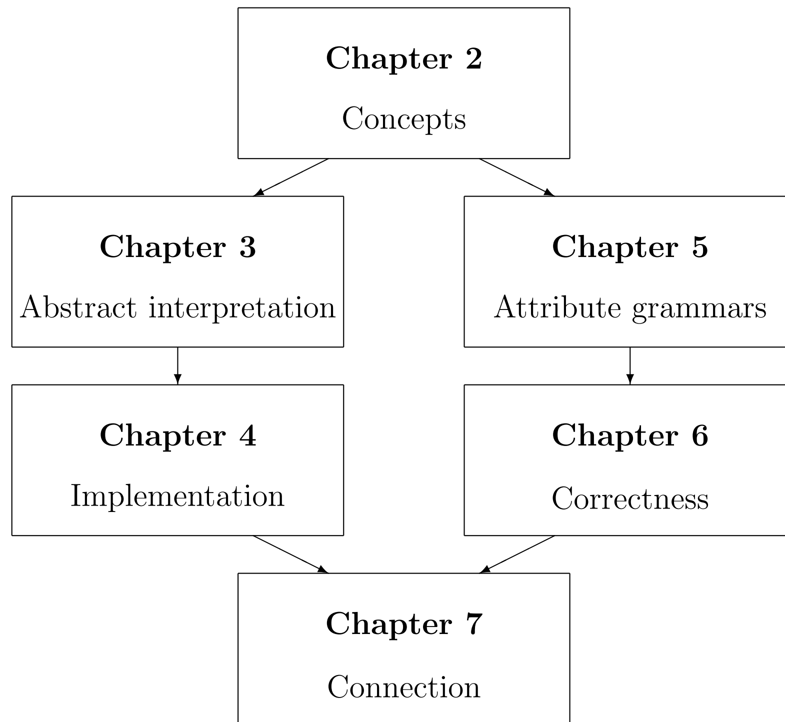
A compiler generator based on attribute grammars is reported in [Paulson 1982] and [Paulson 1984]. Here an extended formalism of attribute grammars (called semantic grammars) is used as a notation for denotational semantics. The generator can transform a semantic grammar into a compiler which produces stack machine code.

Attribute grammar systems are often used as compiler generators. Several systems have been designed so as to produce production quality compilers. Such systems may be seen as being true compiler generators as they from a specification (an attribute grammar) are able to produce compilers. The GAG system [Kastens 1984] and the FNC system [Jourdan, Bellec & Parigot 1990] belong to this category.

1.4 Outline

The central parts of the thesis contain a number of results about abstract interpretation and attribute grammars. These two areas are considered in separate parts before the results are combined.

A simplified plan of dependencies between the chapters of the thesis is illustrated below.



Chapter two This is a general introduction to the theory and concepts used in the rest of the thesis. The central aim in this chapter is to introduce the relevant definitions and theorems about context-free grammars and complete partially ordered sets.

Chapter three and four These chapters consider the specification and implementation of abstract interpretation. In chapter three we describe the basic concepts in abstract interpretation and discuss a number of frameworks used for abstract interpretation. Chapter four is concerned with the implementation of abstract interpretation. The central problem here is to find methods to find fixpoints for functions over a spectrum of domains. A method called *lazy fixpoint iteration* is developed for a class of flat and non-flat domains. The results can both be used for implementing abstract interpretations and for circular attribute grammars.

Chapter five and six In these chapters we consider the specification and proof of attribute grammars. Chapter five gives a general introduction to attribute grammars as a specification language for semantics and data flow analysis. The notion of an attributed scheme is described and its use for proofs of attribute grammars is mentioned. In chapter six we describe a new method for proving attribute grammars correct. The method is based on abstract interpretation in that it uses two different interpretations of an attributed scheme.

Chapter seven An automatic implementation strategy for a class of attribute grammars is described. It combines the fixpoint iteration strategy for abstract interpretation (described in chapter 4) with the proof technique for attribute grammars based on domain-theoretic attribute grammars (described in chapter 6) The framework is illustrated with some examples.

Conclusion The conclusion outlines some directions for future development. A number of other possibilities for combining concepts in abstract interpretation and attribute grammars is mentioned.

Chapter 2

Domains and Grammars

This chapter contains some results from domain theory which will be used in the following chapters. Inclusive relations play an important rôle in our proofs of abstract interpretations and a separate section contains some results on how to lift such relations to recursively defined domains. The final part gives a short introduction to context-free grammars and parse trees. The main purpose of this chapter is to clarify the notation used in later chapters.

Proofs are only included if the results are not found in the standard literature.

2.1 Domain theory

This section introduces the notion of a domain as a complete partially ordered set (cpo) and states the important fixpoint theorem for cpo's. The description in this section follows the treatment in most textbooks. For proofs and further results, please refer to [Schmidt 1986], [Plotkin 1982], and [Wadsworth 1978].

The axiomatic foundation for these concepts is in set theory. We will use λ-notation to describe functions and a number of operations on sets is taken for granted. To start with we will assume the sets $\mathbb{N} = \{\mathsf{0}, \mathsf{1}, \ldots\}$ of natural numbers and $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ of boolean values. The set $\mathbb{A}$ of identifiers contains all the non-empty lists of letters and digits.

2.1.1 Complete partially ordered sets

A *relation* $\mathbf{R}$ between two sets $\mathbf{A}$ and $\mathbf{B}$ is a subset of $\mathbf{A} \times \mathbf{B}$ and we write $\mathcal{R}(\mathbf{A}, \mathbf{B})$ for the set of relations between $\mathbf{A}$ and $\mathbf{B}$. Relationships will normally be written infix, such that:

$$\forall \mathbf{x} \in \mathbf{A}, \mathbf{y} \in \mathbf{B}. \quad \mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow (\mathbf{x}, \mathbf{y}) \in \mathbf{R}$$

A *partial order* is a reflexive, transitive, and antisymmetric relation on a set.

$$\begin{array}{lll} \multicolumn{2}{l}{\sqsubseteq\, \in \mathcal{R}(\mathbf{A},\mathbf{A}) \quad \text{is a partial order}} & \\ \Updownarrow & & \\ \forall\, \mathbf{x} \in \mathbf{A}. & \mathbf{x} \sqsubseteq \mathbf{x} \quad \wedge & \text{—reflexive} \\ \forall\, \mathbf{x},\mathbf{y},\mathbf{z} \in \mathbf{A}. & \mathbf{x} \sqsubseteq \mathbf{y} \wedge \mathbf{y} \sqsubseteq \mathbf{z} \Rightarrow \mathbf{x} \sqsubseteq \mathbf{z} \quad \wedge & \text{—transitive} \\ \forall\, \mathbf{x},\mathbf{y} \in \mathbf{A}. & \mathbf{x} \sqsubseteq \mathbf{y} \wedge \mathbf{y} \sqsubseteq \mathbf{x} \Rightarrow \mathbf{x} = \mathbf{y} & \text{—antisymmetric} \end{array}$$

When convenient the reflected orderings will be used alongside their originals so $\mathbf{x} \sqsupseteq \mathbf{y} \Leftrightarrow \mathbf{y} \sqsubseteq \mathbf{x}$.

A *poset* is tuple $(\mathbf{A}, \sqsubseteq)$ of a set $\mathbf{A}$ and a partial order $\sqsubseteq\, \in \mathcal{R}(\mathbf{A},\mathbf{A})$. We will often refer to posets just by naming their underlying sets and use $\sqsubseteq$ to denote the partial order in a variety of posets. If it is necessary to clarify, the ordering will be indexed with the name of the poset.

A *least element* $\bot_\mathbf{A}$ in a poset $\mathbf{A}$ satisfies

$$\forall\, \mathbf{x} \in \mathbf{A}. \bot_\mathbf{A} \sqsubseteq \mathbf{x}$$

A poset can have at most one least element. The symbol $\bot$ will only be used for least elements and the index will be omitted when the set in question is clear from the context.

A *chain* $\mathbf{c}$ in a poset $\mathbf{A}$ is a function $\mathbf{c} : \mathbb{N} \to \mathbf{A}$ satisfying

$$\mathbf{c}(0) \sqsubseteq \mathbf{c}(1) \sqsubseteq \mathbf{c}(2) \sqsubseteq \cdots$$

We will often use the notation $\langle \mathbf{x_n} \rangle_{\mathbf{n} \geq 0}$ or just $\langle \mathbf{x_n} \rangle_\mathbf{n}$ for the chain $\lambda\, \mathbf{n}.\mathbf{x_n}$.

An *upper bound* $\mathbf{y}$ for a chain $\langle \mathbf{x_n} \rangle_\mathbf{n}$ in a poset $\mathbf{A}$ satisfies

$$\forall\, \mathbf{i} \in \mathbb{N}. \mathbf{x_i} \sqsubseteq \mathbf{y}$$

Upper bounds for chains need not be comparable under the ordering.

A *least upper bound* or *lub* for a chain $\langle \mathbf{x_n} \rangle_\mathbf{n}$ in a poset $\mathbf{A}$ is an upper bound $\mathbf{y}$ which is less than any other upper bound:

$$\forall\, \mathbf{z} \in \mathbf{A}. (\forall\, \mathbf{i} \in \mathbb{N}. \mathbf{x_i} \sqsubseteq \mathbf{z}) \Rightarrow \mathbf{y} \sqsubseteq \mathbf{z}$$

When it exists, the least upper bound of a chain $\langle \mathbf{x_n} \rangle_\mathbf{n}$ will be written as $\bigsqcup_\mathbf{n} \mathbf{x_n}$. A chain can have at most one lub. The least upper bound of a chain is sometimes called the *limit* of the chain.

A *complete partially ordered set* or *cpo* is a poset $\mathbf{A}$ with a least element $\bot_\mathbf{A}$ where any chain has a least upper bound.

A cpo according to this definition is sometimes called a *pointed cpo*, meaning a cpo with a bottom element. It may also be called *chain-complete partial order* as

only chains (and not directed sets) must have least upper bounds. The class of all complete partially ordered sets is called $\mathcal{CPO}$.

A cpo $\mathbf{A}$ is said to be *flat* iff

$$\forall\, \mathbf{x}, \mathbf{y} \in \mathbf{A}.\mathbf{x} \sqsubseteq \mathbf{y} \Rightarrow \mathbf{x} = \bot_{\mathbf{A}} \vee \mathbf{x} = \mathbf{y}$$

and to have *finite height* iff

$$\forall\, \langle \mathbf{x_n} \rangle_{\mathbf{n}}\, \exists\, \mathbf{i} \in \mathbb{N}.\mathbf{x_i} = \bigsqcup_{\mathbf{n}} \mathbf{x_n}$$

A cpo of finite height will have no infinite ascending chains. Such a cpo may, however, have infinite sequences of descending values. An example of this kind is the domain of partially known structures [Rosendahl 1989].

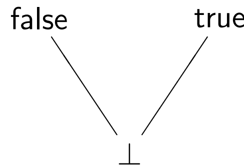
Here, the term *domain* will be used synonymously with cpo. In the literature a *semantic domain* is sometimes defined as an *algebraic cpo*. This is mainly to ensure that the class of domains is closed under various constructions. The term *domain* or *semantic domain* may also be used for bottom-less cpo's [Schmidt 1986]. A further discussion of domains can be found in [Gunter, Mosses & Scott 1989].

Examples The singleton set $\mathbb{U} = \{\bot\}$ with the ordering $\sqsubseteq_{\mathbb{U}} = \{\langle \bot, \bot \rangle\}$ is a cpo. A cpo cannot be empty as it must have a least element.

The cpo $\mathbb{B}_\bot$ of *boolean values* is $\{\bot, \mathsf{true}, \mathsf{false}\}$ with ordering $\sqsubseteq$ such that

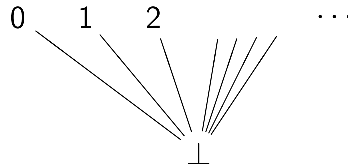
$$\forall\, \mathbf{x} \in \mathbb{B}_\bot.\bot \sqsubseteq \mathbf{x}$$

The ordering can be illustrated as a tree-structure



The cpo $\mathbb{B}_\bot$

The cpo $\mathbb{N}_\bot$ consists of the *natural numbers* $\{0, 1, 2, \ldots\}$ extended with a bottom element $\bot$ with the flat ordering.



The cpo $\mathbb{N}_\bot$

The cpo $\mathbb{A}_\bot$ of *identifiers* or *literals* is the set of character strings extended with a bottom element. $\mathbb{A}_\bot$ is a superset of the identifiers that may occur in actual programming languages.

2.1.2 Fixpoint theorem

For cpo's $\mathbf{A}$ and $\mathbf{B}$, a mapping $\mathbf{f} : \mathbf{A} \rightarrow \mathbf{B}$ is said to be *monotonic* iff

$$\forall \mathbf{x}, \mathbf{y} \in \mathbf{A}. \mathbf{x} \sqsubseteq_{\mathbf{A}} \mathbf{y} \Rightarrow \mathbf{f}(\mathbf{x}) \sqsubseteq_{\mathbf{B}} \mathbf{f}(\mathbf{y})$$

Other names for monotonic maps are *isotone* and *order-preserving*.

For cpo's $\mathbf{A}$ and $\mathbf{B}$ a mapping $\mathbf{f} : \mathbf{A} \rightarrow \mathbf{B}$ is said to be *continuous* iff it is monotonic and

$$\forall \langle \mathbf{x_n} \rangle_{\mathbf{n}} . \mathbf{f}(\bigsqcup \mathbf{x_n}) = \bigsqcup \mathbf{f}(\mathbf{x_n})$$

The monotonicity ensures that $\langle \mathbf{f}(\mathbf{x_n}) \rangle_{\mathbf{n}}$ is a chain in $\mathbf{B}$. Continuous maps are also called *limit-preserving maps*.

A *fixpoint* for a map $\mathbf{f} : \mathbf{A} \rightarrow \mathbf{A}$ is an element $\mathbf{x} \in \mathbf{A}$ such that $\mathbf{x} = \mathbf{f}(\mathbf{x})$.

A mapping $\mathbf{f} : \mathbf{A} \rightarrow \mathbf{B}$ from a poset $\mathbf{A}$ to a poset $\mathbf{B}$ is said to be *strict* iff $\mathbf{f}(\bot_{\mathbf{A}}) = \bot_{\mathbf{B}}$

Fixpoint theorem A continuous map $\mathbf{f} : \mathbf{A} \rightarrow \mathbf{A}$ on a cpo $\mathbf{A}$ has a *least fixpoint* which is the least upper bound of the chain $\langle \mathbf{f^n}(\bot_{\mathbf{A}}) \rangle_{\mathbf{n}}$ and we write

$$\mathsf{fix}(\mathbf{f}) = \bigsqcup_{\mathbf{n}} \mathbf{f^n}(\bot)$$

□

The least fixpoint of a function $\mathbf{f}$ may be written as $\mathsf{lfp}(\mathbf{f})$ to distinguish it from the greatest fixpoint $\mathsf{gfp}(\mathbf{f})$ if that exists.

The chain $\langle \mathbf{f^n}(\bot_{\mathbf{A}}) \rangle_{\mathbf{n}}$ is sometimes called *the ascending Kleene sequence*.

2.1.3 Domain constructors

Definition For two cpo's $(\mathbf{A}, \sqsubseteq_{\mathbf{A}})$ and $(\mathbf{B}, \sqsubseteq_{\mathbf{B}})$ the *Cartesian product* $(\mathbf{A} \times \mathbf{B}, \sqsubseteq_{\mathbf{A} \times \mathbf{B}})$ is defined by:

$$\mathbf{A} \times \mathbf{B} = \{\langle \mathbf{a}, \mathbf{b} \rangle \mid \mathbf{a} \in \mathbf{A} \wedge \mathbf{b} \in \mathbf{B}\}$$
$$\langle \mathbf{a}_1, \mathbf{b}_1 \rangle \sqsubseteq_{\mathbf{A} \times \mathbf{B}} (\mathbf{a}_2, \mathbf{b}_2) \Leftrightarrow \mathbf{a}_1 \sqsubseteq_{\mathbf{A}} \mathbf{a}_2 \wedge \mathbf{b}_1 \sqsubseteq_{\mathbf{B}} \mathbf{b}_2$$

The projections $\mathsf{fst} : \mathbf{A} \times \mathbf{B} \rightarrow \mathbf{A}$ and $\mathsf{snd} : \mathbf{A} \times \mathbf{B} \rightarrow \mathbf{B}$ are for all $\langle \mathbf{x}, \mathbf{y} \rangle \in \mathbf{A} \times \mathbf{B}$,

$$\mathsf{fst}(\langle \mathbf{x}, \mathbf{y} \rangle) = \mathbf{x}$$
$$\mathsf{snd}(\langle \mathbf{x}, \mathbf{y} \rangle) = \mathbf{y}$$

The Cartesian product is sometimes called the lazy product as opposed to the strict (or smashed) product $\{\langle \bot_{\mathbf{A}}, \bot_{\mathbf{B}} \rangle\} \cup (\mathbf{A} \setminus \{\bot_{\mathbf{A}}\}) \times (\mathbf{B} \setminus \{\bot_{\mathbf{B}}\})$.

We will sometimes use a down arrow to denote the selection of elements from a tuple. This means that

$$\langle \mathbf{x}, \mathbf{y} \rangle \downarrow 1 = \mathbf{x} \quad \text{and} \quad \langle \mathbf{x}, \mathbf{y} \rangle \downarrow 2 = \mathbf{y}$$

Infinite products The cartesian product of two cpo's may be generalised to products of a family of cpo's $\mathbf{A_i}, \mathbf{i} \in \mathbf{I}$ for at subset $\mathbf{I} \subseteq \mathbb{N}$ denoted by

$$\prod_{\mathbf{i}\in\mathbf{I}} \mathbf{A_i}$$

We may define a family of functions

$$\mathsf{sel}_\mathbf{j} : \prod_{\mathbf{i}} \mathbf{A_i} \to \mathbf{A_j}, \qquad \mathbf{j} \in \mathbf{I}$$

which selects elements from the tuples of values. The shorter notation $\mathbf{x}{\downarrow}\mathbf{j}$ for $\mathsf{sel}_\mathbf{j}(\mathbf{x})$ may be used when the meaning is clear from the context.

Definition For two posets $(\mathbf{A}, \sqsubseteq_\mathbf{A})$ and $(\mathbf{B}, \sqsubseteq_\mathbf{B})$ the *separated sum* $(\mathbf{A}+\mathbf{B}, \sqsubseteq_{\mathbf{A}+\mathbf{B}})$ can be constructed as

$$\begin{array}{ll} \mathbf{A}+\mathbf{B} = \{\bot_{\mathbf{A}+\mathbf{B}}\} \cup \{\langle 1, \mathbf{a}\rangle \mid \mathbf{a} \in \mathbf{A}\} \cup \{\langle 2, \mathbf{b}\rangle \mid \mathbf{b} \in \mathbf{B}\} & \\ \bot_{\mathbf{A}+\mathbf{B}} \sqsubseteq_{\mathbf{A}+\mathbf{B}} \mathbf{x} & \forall \mathbf{x} \in \mathbf{A}+\mathbf{B} \\ \langle 1, \mathbf{a}_1\rangle \sqsubseteq_{\mathbf{A}+\mathbf{B}} \langle 1, \mathbf{a}_2\rangle & \text{iff } \mathbf{a}_1 \sqsubseteq_\mathbf{A} \mathbf{a}_2 \\ \langle 2, \mathbf{b}_1\rangle \sqsubseteq_{\mathbf{A}+\mathbf{B}} \langle 2, \mathbf{b}_2\rangle & \text{iff } \mathbf{b}_1 \sqsubseteq_\mathbf{B} \mathbf{b}_2 \end{array}$$

We can define the following maps on the sum $\mathbf{A}+\mathbf{B}$.

$$\begin{array}{l} \mathsf{isl}, \mathsf{isr} : \mathbf{A}+\mathbf{B} \to \mathbb{B}_\bot \\ \mathsf{inl} : \mathbf{A} \to \mathbf{A}+\mathbf{B} \\ \mathsf{inr} : \mathbf{B} \to \mathbf{A}+\mathbf{B} \\ \mathsf{outl} : \mathbf{A}+\mathbf{B} \to \mathbf{A} \\ \mathsf{outr} : \mathbf{A}+\mathbf{B} \to \mathbf{B} \end{array}$$

where

$$\begin{array}{lll} \mathsf{isl}(\bot) = \bot & \mathsf{isr}(\bot) = \bot & \\ \mathsf{isl}(\langle 1, \mathbf{x}\rangle) = \mathsf{true} & \mathsf{isr}(\langle 1, \mathbf{x}\rangle) = \mathsf{false} & \mathbf{x} \in \mathbf{A} \\ \mathsf{isl}(\langle 2, \mathbf{x}\rangle) = \mathsf{false} & \mathsf{isr}(\langle 2, \mathbf{x}\rangle) = \mathsf{true} & \mathbf{x} \in \mathbf{B} \\ \mathsf{inl}(\mathbf{x}) = \langle 1, \mathbf{x}\rangle & \mathsf{inr}(\mathbf{x}) = \langle 2, \mathbf{x}\rangle & \\ \mathsf{outl}(\bot) = \bot & \mathsf{outr}(\bot) = \bot & \\ \mathsf{outl}(\langle 1, \mathbf{x}\rangle) = \mathbf{x} & \mathsf{outr}(\langle 1, \mathbf{x}\rangle) = \bot & \mathbf{x} \in \mathbf{A} \\ \mathsf{outl}(\langle 2, \mathbf{x}\rangle) = \bot & \mathsf{outr}(\langle 2, \mathbf{x}\rangle) = \mathbf{x} & \mathbf{x} \in \mathbf{B} \end{array}$$

With $\mathsf{cond} : \mathbb{B}_\bot \times \mathbf{X} \times \mathbf{X} \to \mathbf{X}$ for any cpo $\mathbf{X}$ defined as

$$\begin{array}{ll} \mathsf{cond}(\bot, \mathbf{x}, \mathbf{y}) & = \bot_\mathbf{X} \\ \mathsf{cond}(\mathsf{true}, \mathbf{x}, \mathbf{y}) & = \mathbf{x} \\ \mathsf{cond}(\mathsf{false}, \mathbf{x}, \mathbf{y}) & = \mathbf{y} \end{array}$$

we get

$$\forall \mathbf{x} \in \mathbf{A} + \mathbf{B}.\, \mathsf{cond}(\mathsf{isl}(\mathbf{x}), \mathsf{inl}(\mathsf{outl}(\mathbf{x})), \mathsf{inr}(\mathsf{outr}(\mathbf{x}))) = \mathbf{x}$$

For two cpos $(\mathbf{A}, \sqsubseteq_{\mathbf{A}})$ and $(\mathbf{B}, \sqsubseteq_{\mathbf{B}})$ the *separated sum* $(\mathbf{A} + \mathbf{B}, \sqsubseteq_{\mathbf{A+B}})$ is also a cpo. For two cpos $(\mathbf{A}, \sqsubseteq_{\mathbf{A}})$ and $(\mathbf{B}, \sqsubseteq_{\mathbf{B}})$ the *coalesced sum* $\mathbf{A} \oplus \mathbf{B}$ is $(\mathbf{A} \setminus \{\bot_{\mathbf{A}}\}) + (\mathbf{B} \setminus \{\bot_{\mathbf{B}}\})$.

Infinite sums For a family of cpo's $\mathbf{A_i}, \mathbf{i} \in \mathbf{I}$ for a subset $\mathbf{I} \subseteq \mathbb{N}$ the *infinite sum* is written as

$$\sum_{\mathbf{i}} \mathbf{A_i} = \{\bot_{\Sigma \mathbf{A_i}}\} \cup \{\langle 0, \mathbf{a} \rangle \mid \mathbf{a} \in \mathbf{A}_0\} \cup \cdots$$

$\sum_{\mathbf{i}} \mathbf{A_i}$ is a cpo and we can define a family of functions

$$\mathsf{is_j} : \sum_{\mathbf{i}} \mathbf{A_i} \to \mathbb{B}_\bot$$
$$\mathsf{in_j} : \mathbf{A_j} \to \sum_{\mathbf{i}} \mathbf{A_i}$$
$$\mathsf{out_j} : \sum_{\mathbf{i}} \mathbf{A_i} \to \mathbf{A_j}$$

in the obvious way.

Definition For two cpo's $(\mathbf{A}, \sqsubseteq_{\mathbf{A}})$ and $(\mathbf{B}, \sqsubseteq_{\mathbf{B}})$ the *function domain* $(\mathbf{A} \to \mathbf{B}, \sqsubseteq_{\mathbf{A} \to \mathbf{B}})$ is the cpo of continuous functions from $\mathbf{A}$ to $\mathbf{B}$ with ordering

$$\mathbf{f} \sqsubseteq_{\mathbf{A} \to \mathbf{B}} \mathbf{g} \Leftrightarrow \forall \mathbf{x} \in \mathbf{A}. \mathbf{f}(\mathbf{x}) \sqsubseteq_{\mathbf{B}} \mathbf{g}(\mathbf{x})$$

As a notational convention $\mathbf{f} : \mathbf{A} \to \mathbf{B}$ means that $\mathbf{f}$ is a function from the set $\mathbf{A}$ to the set $\mathbf{B}$ and $\mathbf{f} \in \mathbf{A} \to \mathbf{B}$ means that $\mathbf{f}$ is a continuous function in the domain $\mathbf{A} \to \mathbf{B}$. □

Definition For a set $\mathbf{S}$ the *power set* $\mathbb{P}(\mathbf{S})$ is a cpo with set inclusion as ordering. □

Definition For a cpo $\mathbf{A}$ with least element $\bot_{\mathbf{A}}$ the *lifted* cpo $(\mathbf{A}_\bot, \sqsubseteq_\bot)$ is $\mathbf{A}$ extended with a new bottom element $\bot$ such that

$$\mathbf{A}_\bot = \{\bot\} \cup \{\langle 1, \mathbf{x} \rangle \mid \mathbf{x} \in \mathbf{A}\}$$
$$\bot \sqsubseteq_\bot \mathbf{x} \qquad \forall \mathbf{x} \in \mathbf{A}_\bot$$
$$\langle 1, \mathbf{x} \rangle \sqsubseteq_\bot \langle 1, \mathbf{y} \rangle \quad \Leftrightarrow \mathbf{x} \sqsubseteq \mathbf{y}$$

On a lifted cpo we can define the functions

$$\begin{aligned}
\mathsf{def} &: \mathbf{A}_\perp \to \mathbb{B}_\perp \\
\mathsf{up} &: \mathbf{A} \to \mathbf{A}_\perp \\
\mathsf{down} &: \mathbf{A}_\perp \to \mathbf{A}
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{def}(\perp) &= \perp \\
\mathsf{def}(\langle 1, \mathbf{x} \rangle) &= \mathsf{true} \quad \mathbf{x} \in \mathbf{A} \\
\mathsf{up}(\mathbf{x}) &= \langle 1, \mathbf{x} \rangle \\
\mathsf{down}(\perp) &= \perp_{\mathbf{A}} \\
\mathsf{down}(\langle 1, \mathbf{x} \rangle) &= \mathbf{x}
\end{aligned}$$

These functions satisfy

$$\forall\, \mathbf{x} \in \mathbf{A}_\perp.\, \mathsf{cond}(\mathsf{def}(\mathbf{x}), \mathsf{up}(\mathsf{down}(\mathbf{x})), \perp) = \mathbf{x}$$

Finite lists For a cpo $(\mathbf{A}, \sqsubseteq)$ the cpo of *finite lists* of values from A is the set $\mathbf{A}^*$:

$$\mathbf{A}^* = \mathbb{U} + \mathbf{A} + (\mathbf{A} \times \mathbf{A}) + (\mathbf{A} \times \mathbf{A} \times \mathbf{A}) + \cdots$$

Infinite lists For a cpo $(\mathbf{A}, \sqsubseteq)$ the cpo of *infinite lists* of values from A is the set $\mathsf{List}\,\mathbf{A}$:

$$\mathsf{List}\,\mathbf{A} = \prod_{\mathbf{i}} \mathbf{A}$$

The bottom element in this domain is the infinite list $[\perp_{\mathbf{A}}, \perp_{\mathbf{A}}, \ldots]$. The cpo of infinite list may also be defined as

$$\mathsf{List}\,\mathbf{A} = \mathbb{N} \to \mathbf{A}$$

or as the solution to the recursive domain equation

$$\mathbf{D} = \mathbf{A} \times \mathbf{D}$$

Externally these definitions are identical.

Domain expressions A *domain expression* $\mathbf{F}(\mathbf{X})$ in the variable $\mathbf{X}$ is a string in the language

$$\begin{aligned}
\langle \mathbf{exp} \rangle \ ::=\ & \mathbf{C_i} \mid \mathbf{X} \\
\mid\ & (\langle \mathbf{exp} \rangle \to \langle \mathbf{exp} \rangle) \\
\mid\ & (\langle \mathbf{exp} \rangle \times \langle \mathbf{exp} \rangle) \\
\mid\ & (\langle \mathbf{exp} \rangle + \langle \mathbf{exp} \rangle) \\
\mid\ & \langle \mathbf{exp} \rangle_\perp
\end{aligned}$$

where $\mathbf{C_i}, \mathbf{i} \geq 0$ are names for cpo's. For any cpo $\mathbf{A}$, $\mathbf{F}(\mathbf{A})$ denotes the cpo obtained by substituting $\mathbf{A}$ for any occurrence of $\mathbf{X}$ in $\mathbf{F}(\mathbf{X})$ and interpreting the resulting expression as a cpo specification.

2.1.4 Recursive domain equations

For a domain expression $\mathbf{F}(\mathbf{X})$ it is possible to define a cpo $\mathbf{D}_\infty$ such that $\mathbf{D}_\infty$ is isomorphic to $\mathbf{F}(\mathbf{D}_\infty)$. The cpo will also be denoted $\mathsf{rec}\ \mathbf{X}.\mathbf{F}(\mathbf{X})$ or we may write the domain equation as $\mathbf{D}_\infty \cong \mathbf{F}(\mathbf{D}_\infty)$.

Definition A cpo $\mathbf{A}$ can be *embedded* in a cpo $\mathbf{B}$ if there exists a *retraction-pair* $\langle \mathbf{i}, \mathbf{j} \rangle$ where $\mathbf{i} \in \mathbf{A} \to \mathbf{B}$ and $\mathbf{j} \in \mathbf{B} \to \mathbf{A}$ are continuous functions, such that

$$\begin{aligned} \mathbf{j} \circ \mathbf{i} &= \mathsf{id}_\mathbf{A} \\ \mathbf{i} \circ \mathbf{j} &\sqsubseteq \mathsf{id}_\mathbf{B} \end{aligned}$$

A retraction pair will often be illustrated as

$$\mathbf{A} \underset{\mathbf{j}}{\overset{\mathbf{i}}{\rightleftarrows}} \mathbf{B}$$

and we write its type as $\langle \mathbf{i}, \mathbf{j} \rangle : \mathbf{A} \leftrightarrow \mathbf{B}$.

The function $\mathbf{i}$ is called an injection or an *embedding* and $\mathbf{j}$ is called a *projection*.

Retraction pairs can be composed in the obvious fashion. For

$$\begin{aligned} &\langle \mathbf{i}, \mathbf{j} \rangle : \mathbf{A} \leftrightarrow \mathbf{B} \\ &\langle \mathbf{k}, \mathbf{l} \rangle : \mathbf{B} \leftrightarrow \mathbf{C} \end{aligned}$$

the pair

$$\langle \mathbf{k}, \mathbf{l} \rangle \circ \langle \mathbf{i}, \mathbf{j} \rangle = \langle \mathbf{k} \circ \mathbf{i}, \mathbf{j} \circ \mathbf{l} \rangle : \mathbf{A} \leftrightarrow \mathbf{C}$$

is a retraction pair.

Lemma Retraction pairs (and generally pairs of continuous functions) can be lifted through domain constructions as follows. Let $\langle \mathbf{f}, \mathbf{g} \rangle : \mathbf{A} \leftrightarrow \mathbf{B}$ and $\langle \mathbf{f}', \mathbf{g}' \rangle : \mathbf{A}' \leftrightarrow \mathbf{B}'$ be retraction pairs.

$$\begin{aligned} &\langle \mathbf{f}, \mathbf{g} \rangle \times \langle \mathbf{f}', \mathbf{g}' \rangle : \mathbf{A} \times \mathbf{A}' \leftrightarrow \mathbf{B} \times \mathbf{B}' \\ &\langle \mathbf{f}, \mathbf{g} \rangle + \langle \mathbf{f}', \mathbf{g}' \rangle : \mathbf{A} + \mathbf{A}' \leftrightarrow \mathbf{B} + \mathbf{B}' \\ &\langle \mathbf{f}, \mathbf{g} \rangle \to \langle \mathbf{f}', \mathbf{g}' \rangle : (\mathbf{A} \to \mathbf{A}') \leftrightarrow (\mathbf{B} \to \mathbf{B}') \\ &\langle \mathbf{f}, \mathbf{g} \rangle_\bot : \mathbf{A}_\bot \leftrightarrow \mathbf{B}_\bot \end{aligned}$$

defined by

$$\begin{aligned} \langle \mathbf{f}, \mathbf{g} \rangle \times \langle \mathbf{f}', \mathbf{g}' \rangle &= \langle \lambda\,\mathbf{x}.\, \langle \mathbf{f}(\mathsf{fst}(\mathbf{x})), \mathbf{f}'(\mathsf{snd}(\mathbf{x})) \rangle, \lambda\,\mathbf{y}.\, \langle \mathbf{g}(\mathsf{fst}(\mathbf{y})), \mathbf{g}'(\mathsf{snd}(\mathbf{y})) \rangle \rangle \\ \langle \mathbf{f}, \mathbf{g} \rangle + \langle \mathbf{f}', \mathbf{g}' \rangle &= \langle\ \lambda\,\mathbf{x} \in \mathbf{A} + \mathbf{A}'.\, \mathsf{cond}(\mathsf{isl}(\mathbf{x}), \mathsf{inl}(\mathbf{f}(\mathsf{outl}(\mathbf{x}))), \mathsf{inr}(\mathbf{f}'(\mathsf{outr}(\mathbf{x})))), \\ &\qquad \lambda\,\mathbf{x} \in \mathbf{B} + \mathbf{B}'.\, \mathsf{cond}(\mathsf{isl}(\mathbf{x}), \mathsf{inl}(\mathbf{g}(\mathsf{outl}(\mathbf{x}))), \mathsf{inr}(\mathbf{g}(\mathsf{outr}(\mathbf{x})))) \rangle \\ \langle \mathbf{f}, \mathbf{g} \rangle \to \langle \mathbf{f}', \mathbf{g}' \rangle &= \langle \lambda\,\mathbf{x} \in \mathbf{A} \to \mathbf{A}'.(\mathbf{f}' \circ \mathbf{x} \circ \mathbf{g}), \lambda\,\mathbf{x} \in \mathbf{B} \to \mathbf{B}'.(\mathbf{g}' \circ \mathbf{x} \circ \mathbf{f}) \rangle \\ \langle \mathbf{f}, \mathbf{g} \rangle_\bot : \mathbf{A}_\bot \leftrightarrow \mathbf{B}_\bot &= \langle\ \lambda\,\mathbf{x}.\, \mathsf{cond}(\mathsf{def}(\mathbf{x}), \mathsf{up}(\mathbf{f}(\mathsf{down}(\mathbf{x}))), \bot), \\ &\qquad \lambda\,\mathbf{x}.\, \mathsf{cond}(\mathsf{def}(\mathbf{x}), \mathsf{up}(\mathbf{g}(\mathsf{down}(\mathbf{x}))), \bot) \rangle \end{aligned}$$

For any domain $\mathbf{C}$ the pair $\langle \mathsf{id}_\mathbf{C}, \mathsf{id}_\mathbf{C} \rangle : \mathbf{C} \leftrightarrow \mathbf{C}$ is a retraction pair.

Lemma Let $\mathbf{F}(\mathbf{X})$ be a domain expression for any domain $\mathbf{X}$. For a retraction pair $\langle \mathbf{f}, \mathbf{g} \rangle : \mathbf{A} \leftrightarrow \mathbf{B}$ we can define a retraction pair $\mathbf{F}\langle \mathbf{f}, \mathbf{g} \rangle : \mathbf{F}(\mathbf{A}) \leftrightarrow \mathbf{F}(\mathbf{B})$.

Warning It is not always possible to embed a cpo $\mathbf{A}$ in $\mathbf{F}(\mathbf{A})$ as $\mathbb{N}_\perp$ cannot be embedded in $\mathbb{N}_\perp \rightarrow \mathbb{U}$.

Lemma For $\langle \mathbf{f}, \mathbf{g} \rangle : \mathbf{A} \leftrightarrow \mathbf{B}$ and $\langle \mathbf{f}', \mathbf{g}' \rangle : \mathbf{B} \leftrightarrow \mathbf{C}$,

$$\mathbf{F}\langle \mathbf{f}' \circ \mathbf{f}, \mathbf{g} \circ \mathbf{g}' \rangle = \mathbf{F}\langle \mathbf{f}', \mathbf{g}' \rangle \circ \mathbf{F}\langle \mathbf{f}, \mathbf{g} \rangle$$

Lemma Let $\mathbb{U}$ be the one-point cpo $\{\perp\}$ as defined earlier. For any domain expression $\mathbf{F}(\mathbf{X})$, $\mathbb{U}$ can be embedded in $\mathbf{F}(\mathbb{U})$ by

$$\langle \phi_0, \psi_0 \rangle : \mathbb{U} \leftrightarrow \mathbf{F}(\mathbb{U})$$

where

$$\phi_0 = \lambda\, \mathbf{x}.\perp_{\mathbf{F}(\mathbb{U})}$$

and

$$\psi_0 = \lambda\, \mathbf{x}.\perp_{\mathbb{U}}$$

Definition For a domain expression $\mathbf{F}$, define the sequence of cpo's $\langle \mathbf{D_i} \rangle_{\mathbf{i} \geq 0}$ as

$$\mathbf{D}_0 = \mathbb{U}$$
$$\mathbf{D_i} = \mathbf{F^n}(\mathbb{U})$$

$\mathbf{D_i}$ can be embedded in $\mathbf{D_{i+1}}$ by the retraction pair

$$\langle \phi_\mathbf{i}, \psi_\mathbf{i} \rangle = \mathbf{F^i}\langle \phi_0, \psi_0 \rangle$$

with type

$$\langle \phi_\mathbf{i}, \psi_\mathbf{i} \rangle : \mathbf{D_i} \leftrightarrow \mathbf{D_{i+1}}$$

$$\mathbf{D_i} \underset{\psi_\mathbf{j}}{\overset{\phi_\mathbf{i}}{\rightleftarrows}} \mathbf{D_{i+1}}$$

Definition The *inverse limit* of the sequence $\langle \mathbf{D_i} \rangle_{\mathbf{i} \geq 0}$ is $\mathbf{D}_\infty$ where

$$\mathbf{D}_\infty = \{[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x_i}, \ldots] \mid \forall\, \mathbf{n}.\mathbf{x_n} \in \mathbf{D_n} \wedge \mathbf{x_n} = \psi_\mathbf{n}(\mathbf{x_{n+1}})\}$$

Theorem $\mathbf{D}_\infty$ is a cpo with ordering

$$[\mathbf{x}_0, \mathbf{x}_1, \ldots] \sqsubseteq [\mathbf{y}_0, \mathbf{y}_1, \ldots] \Leftrightarrow \forall\, \mathbf{n}.\mathbf{x}_\mathbf{n} \sqsubseteq_{\mathbf{D}_\mathbf{n}} \mathbf{y}_\mathbf{n}$$

Theorem $\mathbf{D}_\mathbf{m}$ can be embedded in $\mathbf{D}_\infty$ by the retraction pair $\langle \theta_{\mathbf{m}\infty}, \theta_{\infty\mathbf{m}} \rangle$ where

$$\begin{aligned}\theta_{\mathbf{m}\infty} = \lambda\, \mathbf{x} \in \mathbf{D}_\mathbf{m}.[\, &\psi_0 \circ \cdots \circ \psi_{\mathbf{m}-1}(\mathbf{x}), \psi_1 \circ \cdots \circ \psi_{\mathbf{m}-1}(\mathbf{x}), \ldots, \\ &\psi_{\mathbf{m}-1}(\mathbf{x}), \mathbf{x}, \phi_\mathbf{m}(\mathbf{x}), \phi_{\mathbf{n}+1} \circ \phi_\mathbf{m}(\mathbf{x}), \ldots]\end{aligned}$$

and

$$\theta_{\infty\mathbf{m}} = \lambda\, \mathbf{x} \in \mathbf{D}_\infty.\mathbf{x}{\downarrow}\mathbf{m}$$

Lemma The retraction pair $\langle \theta_{\mathbf{m}\infty}, \theta_{\infty\mathbf{m}} \rangle : \mathbf{D}_\mathbf{m} \leftrightarrow \mathbf{D}_\infty$ satisfy

$$\langle \theta_{\mathbf{m}\infty}, \theta_{\infty\mathbf{m}} \rangle = \langle \theta_{\mathbf{m}+1\infty}, \theta_{\infty\mathbf{m}+1} \rangle \circ \langle \phi_\mathbf{m}, \psi_\mathbf{m} \rangle$$

Corollary $\mathbf{D}_{\mathbf{m}+1}$ can be embedded in $\mathbf{F}(\mathbf{D}_\infty)$ by the retraction pair $\mathbf{F}\,\langle \theta_{\mathbf{m}\infty}, \theta_{\infty\mathbf{m}} \rangle$

Definition Write $\langle \theta'_{\mathbf{m}\infty}, \theta'_{\infty\mathbf{m}} \rangle$ for $\mathbf{F}\,\langle \theta_{\mathbf{m}-1\infty}, \theta_{\infty\mathbf{m}-1} \rangle$ and it follows that

$$\langle \theta'_{\mathbf{m}\infty}, \theta'_{\infty\mathbf{m}} \rangle = \langle \theta'_{\mathbf{m}+1\infty}, \theta'_{\infty\mathbf{m}+1} \rangle \circ \langle \phi_\mathbf{m}, \psi_\mathbf{m} \rangle$$

Definition Let $\langle \Phi, \Psi \rangle : \mathbf{D}_\infty \leftrightarrow \mathbf{F}(\mathbf{D}_\infty)$ be defined as

$$\Phi = \lambda\, \mathbf{x}. \bigsqcup_{\mathbf{m}=1}^{\infty} \theta'_{\mathbf{m}\infty}(\theta_{\infty\mathbf{m}}(\mathbf{x}))$$

$$\Psi = \lambda\, \mathbf{x}. \bigsqcup_{\mathbf{m}=1}^{\infty} \theta_{\mathbf{m}\infty}(\theta'_{\infty\mathbf{m}}(\mathbf{x}))$$

The functionality of the various functions can be illustrated with this diagram:

$$\begin{array}{ccc} \mathbf{D}_\mathbf{m} & \underset{\theta_{\infty\mathbf{m}}}{\overset{\theta_{\mathbf{m}\infty}}{\rightleftarrows}} & \mathbf{D}_\infty \\ \phi_\mathbf{m} \downarrow \uparrow \psi_\mathbf{m} & & \Phi \downarrow \uparrow \Psi \\ \mathbf{D}_{\mathbf{m}+1} & \underset{\theta'_{\infty\mathbf{m}+1}}{\overset{\theta'_{\mathbf{m}+1\infty}}{\rightleftarrows}} & \mathbf{F}(\mathbf{D}_\infty) \end{array}$$

Theorem

$$\Phi \circ \Psi = \mathsf{id}_{\mathbf{F}(\mathbf{D}_\infty)} \quad \text{and} \quad \Psi \circ \Phi = \mathsf{id}_{\mathbf{D}_\infty}$$

□

The theorem says that $\mathbf{F}(\mathbf{D}_\infty)$ and $\mathbf{D}_\infty$ are isomorphic. This means that elements in $\mathbf{D}_\infty$ may be interpreted as belonging to $\mathbf{F}(\mathbf{D}_\infty)$ using the proper injection function (Φ) and *vice versa* (using Ψ).

2.1.5 Lattices

A *complete lattice* $(\mathbf{A}, \sqsubseteq, \bigsqcup, \bigsqcap)$ is a poset $(\mathbf{A}, \sqsubseteq)$ where every subset $\mathbf{X} \subseteq \mathbf{A}$ has a least upper bound $\bigsqcup \mathbf{X}$ and a greatest lower bound $\bigsqcap \mathbf{X}$.

$$\forall\, \mathbf{y} \in \mathbf{A}.(\forall\, \mathbf{x} \in \mathbf{X}.\mathbf{x} \sqsubseteq \mathbf{y}) \Rightarrow \bigsqcup \mathbf{X} \sqsubseteq \mathbf{y}$$
$$\forall\, \mathbf{y} \in \mathbf{A}.(\forall\, \mathbf{x} \in \mathbf{X}.\mathbf{y} \sqsubseteq \mathbf{x}) \Rightarrow \mathbf{y} \sqsubseteq \bigsqcap \mathbf{X}$$

A complete lattice $(\mathbf{A}, \sqsubseteq, \bigsqcup, \bigsqcap)$ is also a cpo $(\mathbf{A}, \sqsubseteq)$ with least element $\bigsqcap \mathbf{A}$ and top element $\bigsqcup \mathbf{A}$.

For any set $\mathbf{S}$ the powerset $\mathbb{P}(\mathbf{S})$ is a complete lattice.

A monotone function $\mathbf{f} : \mathbf{A} \to \mathbf{A}$ on a complete lattice $\mathbf{A}$ will have a greatest fixpoint $\mathsf{gfp}(\mathbf{f})$ and a least fixpoint $\mathsf{lfp}(\mathbf{f})$ [Tarski 1955]. These fixpoints can be found as:

$$\mathsf{lfp}(\mathbf{f}) = \bigsqcap \{\mathbf{x} \in \mathbf{A} \mid \mathbf{f}(\mathbf{x}) \sqsubseteq \mathbf{x}\}$$
$$\mathsf{gfp}(\mathbf{f}) = \bigsqcup \{\mathbf{x} \in \mathbf{A} \mid \mathbf{f}(\mathbf{x}) \sqsupseteq \mathbf{x}\}$$

It should be noticed that the limit of the ascending Kleene sequence with a monotonic (but not continuous) function will not necessarily be the least fixpoint or even a fixpoint at all.

2.2 Inclusive relations

This section introduces the notion of *inclusive relations* and defines the extension of relations on base types over a type structure. Most of the results can be found in [Reynolds 1983], [Plotkin 1982], and [Manna, Ness & Vuillemin 1972]. The extension to recursively defined domains has been made in a category-theoretic framework in [Smyth & Plotkin 1982]. In this section we provide a construction based on the inverse limits as used in the last section.

Definition A relation $\mathbf{R} \in \mathcal{R}(\mathbf{A}, \mathbf{B})$ between two cpo's $\mathbf{A}$ and $\mathbf{B}$ is said to be *inclusive* iff

$$\forall \langle \mathbf{x_n} \rangle_\mathbf{n}, \langle \mathbf{y_n} \rangle_\mathbf{n} . (\forall \mathbf{i} \in \mathbb{N}. \mathbf{x_i} \; \mathbf{R} \; \mathbf{y_i}) \Rightarrow \bigsqcup_\mathbf{n} \mathbf{x_n} \; \mathbf{R} \bigsqcup_\mathbf{n} \mathbf{y_n}$$

Inclusive predicates are sometimes called *admissible*, *inductive*, or *complete*. The class of inclusive relations between cpo's $\mathbf{A}$ and $\mathbf{B}$ is denoted $\mathcal{R}_\mathbf{i}(\mathbf{A}, \mathbf{B})$.

A relation $\mathbf{R} \in \mathcal{R}(\mathbf{A}, \mathbf{B})$ is said to be *strict* iff

$$\perp_\mathbf{A} \mathbf{R} \perp_\mathbf{B}$$

Fixpoint induction Let $\mathbf{R} \in \mathcal{R}(\mathbf{A}, \mathbf{B})$ be a strict and inclusive relation between cpo's and $\mathbf{f} : \mathbf{A} \to \mathbf{A}$ and $\mathbf{g} : \mathbf{B} \to \mathbf{B}$ be continuous functions such that

$$\forall \mathbf{a} \in \mathbf{A}, \mathbf{b} \in \mathbf{B}. \mathbf{a} \; \mathbf{R} \; \mathbf{b} \Rightarrow \mathbf{f}(\mathbf{a}) \; \mathbf{R} \; \mathbf{g}(\mathbf{b})$$

The fixpoints of $\mathbf{f}$ and $\mathbf{g}$ will then also be related by $\mathbf{R}$: $\mathsf{fix}(\mathbf{f}) \; \mathbf{R} \; \mathsf{fix}(\mathbf{g})$. $\square$

2.2.1 Logical relations

The extension of relations from base domains to composite domains is often referred to as *logical*. We will sometimes use $\trianglelefteq$ to denote any inclusive relation extended from a relation $\trianglelefteq$ on the base domains through a type structure.

Lemma If $\mathbf{R} \in \mathcal{R}_\mathbf{i}(\mathbf{A}, \mathbf{B})$ and $\mathbf{S} \in \mathcal{R}_\mathbf{i}(\mathbf{A}, \mathbf{B})$ then so are $\mathbf{R} \cup \mathbf{S}$ and $\mathbf{R} \cap \mathbf{S}$. The sets $\varnothing$, $\{\langle \perp_\mathbf{A}, \perp_\mathbf{B} \rangle\}$, and $\mathbf{A} \times \mathbf{B}$ are also inclusive.

Warning If $\mathbf{R} \in \mathcal{R}_\mathbf{i}(\mathbf{A}, \mathbf{B})$ and $\mathbf{S} \in \mathcal{R}_\mathbf{i}(\mathbf{A}, \mathbf{B})$ then $\mathbf{R} \setminus \mathbf{S}$ is not always inclusive and if $\mathbf{R} \in \mathcal{R}_\mathbf{i}(\mathbf{A}, \mathbf{B})$ and $\mathbf{S} \in \mathcal{R}_\mathbf{i}(\mathbf{B}, \mathbf{C})$ then their composition

$$(\mathbf{S} \circ \mathbf{R}) = \{\langle \mathbf{a}, \mathbf{c} \rangle \in \mathbf{A} \times \mathbf{C} \mid \exists \mathbf{b} \in \mathbf{B}. \mathbf{a} \; \mathbf{R} \; \mathbf{b} \wedge \mathbf{b} \; \mathbf{S} \; \mathbf{c}\}$$

need not be inclusive.

For any continuous function $\mathbf{f} : \mathbf{A} \to \mathbf{B}$ we can define the inclusive relations

$$\{\langle \mathbf{a}, \mathbf{b} \rangle \mid \mathbf{f}(\mathbf{a}) \sqsubseteq \mathbf{b}\}$$
$$\{\langle \mathbf{a}, \mathbf{b} \rangle \mid \mathbf{f}(\mathbf{a}) \sqsupseteq \mathbf{b}\}$$
$$\{\langle \mathbf{a}, \mathbf{b} \rangle \mid \mathbf{f}(\mathbf{a}) = \mathbf{b}\}$$

More generally we have the following theorem [Manna, Ness & Vuillemin 1972]:

Theorem A relation $\mathbf{R}$ is inclusive if it has the form

$$\mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow \langle \mathbf{IR} \rangle$$

where $\langle \mathbf{IR} \rangle$ is an expression in $\mathbf{x}$ and $\mathbf{y}$ of the form

$$\begin{array}{lcl} \langle \mathbf{IR} \rangle & ::= & \langle \mathbf{IR} \rangle \wedge \langle \mathbf{IR} \rangle \mid \forall\, \mathbf{d_i} \in \mathbf{D_i} : \langle \mathbf{ER} \rangle \\ \langle \mathbf{ER} \rangle & ::= & \langle \mathbf{ER} \rangle \vee \langle \mathbf{ER} \rangle \mid \mathbf{Q_j}(\mathbf{d_i}) \mid \mathbf{f}(\mathbf{x}, \mathbf{y}, \mathbf{d_i}) \sqsubseteq \mathbf{g}(\mathbf{x}, \mathbf{y}, \mathbf{d_i}) \end{array}$$

where $\mathbf{Q_j}$ is a first-order predicate and $\mathbf{f}$ and $\mathbf{g}$ are two continuous functions.

Definition Let $\mathbf{R} \in \mathcal{R}_\mathbf{i}(\mathbf{A}, \mathbf{A}')$ and $\mathbf{S} \in \mathcal{R}_\mathbf{i}(\mathbf{B}, \mathbf{B}')$ be inclusive relations. The following relations are also inclusive:

$$\begin{array}{ll} \mathbf{R} \boxtimes \mathbf{S} & \in \mathcal{R}_\mathbf{i}(\mathbf{A} \times \mathbf{B}, \mathbf{A}' \times \mathbf{B}') \\ \mathbf{R} \boxplus \mathbf{S} & \in \mathcal{R}_\mathbf{i}(\mathbf{A} + \mathbf{B}, \mathbf{A}' + \mathbf{B}') \\ \mathbf{R} \boxed{\rightarrow} \mathbf{S} & \in \mathcal{R}_\mathbf{i}(\mathbf{A} \rightarrow \mathbf{B}, \mathbf{A}' \rightarrow \mathbf{B}') \\ \boxdot \mathbf{R} & \in \mathcal{R}_\mathbf{i}(\mathbf{A}_\perp, \mathbf{A}'_\perp) \end{array}$$

defined by

$$\begin{array}{lcl} \mathbf{x}\ \mathbf{R} \boxtimes \mathbf{S}\ \mathbf{y} & \Leftrightarrow & \mathsf{fst}(\mathbf{x})\ \mathbf{R}\ \mathsf{fst}(\mathbf{y}) \wedge \mathsf{snd}(\mathbf{x})\ \mathbf{S}\ \mathsf{snd}(\mathbf{y}) \\ \mathbf{x}\ \mathbf{R} \boxplus \mathbf{S}\ \mathbf{y} & \Leftrightarrow & (\mathbf{x} = \perp \wedge \mathbf{y} = \perp) \vee \\ & & (\mathsf{isl}(\mathbf{x}) \wedge \mathsf{isl}(\mathbf{y}) \wedge \mathsf{outl}(\mathbf{x})\ \mathbf{R}\ \mathsf{outl}(\mathbf{y})) \vee \\ & & (\mathsf{isr}(\mathbf{x}) \wedge \mathsf{isr}(\mathbf{y}) \wedge \mathsf{outr}(\mathbf{x})\ \mathbf{S}\ \mathsf{outr}(\mathbf{y})) \\ \mathbf{x}\ \mathbf{R} \boxed{\rightarrow} \mathbf{S}\ \mathbf{y} & \Leftrightarrow & \forall\, \mathbf{a} \in \mathbf{A}, \mathbf{a}' \in \mathbf{A}'.\mathbf{a}\ \mathbf{R}\ \mathbf{a}' \Rightarrow \mathbf{x}(\mathbf{a})\ \mathbf{S}\ \mathbf{y}(\mathbf{a}') \\ \mathbf{x} \boxdot \mathbf{R}\ \mathbf{y} & \Leftrightarrow & (\mathbf{x} = \perp \wedge \mathbf{y} = \perp) \vee \\ & & (\mathbf{x} \neq \perp \wedge \mathbf{y} \neq \perp \wedge \mathsf{down}(\mathbf{x})\ \mathbf{R}\ \mathsf{down}(\mathbf{y})) \end{array}$$

For any cpo $\mathbf{C}$ the equality relation

$$\Delta_\mathbf{C} \in \mathcal{R}_\mathbf{i}(\mathbf{C}, \mathbf{C})$$

defined by

$$\mathbf{x}\ \Delta_\mathbf{C}\ \mathbf{y} \Leftrightarrow \mathbf{x} = \mathbf{y}$$

is inclusive.

Domain expressions Let $\mathbf{A}$ and $\mathbf{B}$ be cpos with an inclusive relation $\mathbf{R} \in \mathcal{R}_\mathbf{i}(\mathbf{A}, \mathbf{B})$. For a domain equation $\mathbf{F}(\mathbf{X})$ in $\mathbf{X}$ we can then define an inclusive relation $\boxed{\mathrm{F}}\mathbf{R} \in \mathcal{R}_\mathbf{i}(\mathbf{F}(\mathbf{A}), \mathbf{F}(\mathbf{B}))$ by structural induction over the definition of $\mathbf{F}$ using the above.

Comment The rules above for extending relations to composite cpo's are not the only possible definitions. [Mycroft & Jones 1986] give two other extensions for sum domains but these extensions are not strict and they can therefore not be used for fixpoint induction.

2.2.2 Inclusive relations on recursive domains

It is possible to lift inclusive relations to recursively defined cpo's.

Theorem Let $\mathbf{F}(\mathbf{X},\mathbf{Y})$ be a domain expression in $\mathbf{X}$ and $\mathbf{Y}$ and $\mathbf{R} \in \mathcal{R}_\mathbf{i}(\mathbf{A},\mathbf{B})$. Let $\mathbf{D}^1 = \mathsf{rec}\ \mathbf{Y}.\mathbf{F}(\mathbf{A},\mathbf{Y})$ and $\mathbf{D}^2 = \mathsf{rec}\ \mathbf{Y}.\mathbf{F}(\mathbf{B},\mathbf{Y})$, then $\mathbf{R}$ can be extended to an inclusive relation $\mathbf{R}_\infty \in \mathcal{R}_\mathbf{i}(\mathbf{D}^1,\mathbf{D}^2)$.

Construction Let $\mathbf{G}(\mathbf{Y}) = \mathbf{F}(\mathbf{A},\mathbf{Y})$ and $\mathbf{H}(\mathbf{Y}) = \mathbf{F}(\mathbf{B},\mathbf{Y})$. In the construction of $\mathbf{D}^1$ and $\mathbf{D}^2$ we define

$$\mathbf{D}^1_\mathbf{m} = \mathbf{G}^\mathbf{m}(\mathbb{U}) \qquad \text{and} \qquad \mathbf{D}^2_\mathbf{m} = \mathbf{H}^\mathbf{m}(\mathbb{U})$$

and we can define the relations

$$\mathbf{R}_\mathbf{m} \in \mathcal{R}_\mathbf{i}(\mathbf{D}^1_\mathbf{m},\mathbf{D}^2_\mathbf{m})$$

by

$$\begin{aligned}\mathbf{R}_0 &= \{\langle\bot,\bot\rangle\}\\ \mathbf{R}_{\mathbf{m}+1} &= \mathbb{F}(\mathbf{R},\mathbf{R}_\mathbf{m})\end{aligned}$$

The relation $\mathbf{R}_\infty \in \mathcal{R}_\mathbf{i}(\mathbf{D}^1,\mathbf{D}^2)$ is defined as

$$\mathbf{x}\ \mathbf{R}_\infty\ \mathbf{y} \Leftrightarrow \forall\,\mathbf{m}.\mathbf{x}{\downarrow}\mathbf{m}\ \mathbf{R}_\mathbf{m}\ \mathbf{y}{\downarrow}\mathbf{m}$$

where the phrase $\mathbf{x}{\downarrow}\mathbf{m}$ denotes the selection of the $\mathbf{m}^{\text{th}}$ element in the vector $\mathbf{x}$.

Proof $\mathbf{R}_\infty$ is inclusive:
Let $\langle\mathbf{a}_\mathbf{n}\rangle_{\mathbf{n}\geq 0}$ and $\langle\mathbf{b}_\mathbf{n}\rangle_{\mathbf{n}\geq 0}$ be chains in $\mathbf{D}^1$ and $\mathbf{D}^2$ such that

$$\forall\,\mathbf{m}.\mathbf{a}_\mathbf{m}\ \mathbf{R}_\infty\ \mathbf{b}_\mathbf{m}$$

We want to show that

$$\bigsqcup_\mathbf{i}\mathbf{a}_\mathbf{i}\ \mathbf{R}_\infty \bigsqcup_\mathbf{i}\mathbf{b}_\mathbf{i}$$

Lubs in $\mathbf{D}^1$ and $\mathbf{D}^2$ are found componentwise, hence

$$\begin{aligned}\bigsqcup_\mathbf{i}\mathbf{a}_\mathbf{i} &= [\bigsqcup_\mathbf{i}(\mathbf{a}_\mathbf{i}{\downarrow}0),\bigsqcup_\mathbf{i}(\mathbf{a}_\mathbf{i}{\downarrow}1),\ldots]\\ \bigsqcup_\mathbf{i}\mathbf{b}_\mathbf{i} &= [\bigsqcup_\mathbf{i}(\mathbf{b}_\mathbf{i}{\downarrow}0),\bigsqcup_\mathbf{i}(\mathbf{b}_\mathbf{i}{\downarrow}1),\ldots]\end{aligned}$$

and by definition

$$\forall\,\mathbf{i}\,\forall\,\mathbf{m}.(\mathbf{a}_\mathbf{i}{\downarrow}\mathbf{m})\ \mathbf{R}_\mathbf{m}\ (\mathbf{b}_\mathbf{i}{\downarrow}\mathbf{m})$$

$\mathbf{R}_\mathbf{m}$ is inclusive for any $\mathbf{m}$ so

$$\forall\,\mathbf{m}.\bigsqcup_\mathbf{i}(\mathbf{a}_\mathbf{i}{\downarrow}\mathbf{m})\ \mathbf{R}_\mathbf{m} \bigsqcup_\mathbf{i}(\mathbf{b}_\mathbf{i}{\downarrow}\mathbf{m})$$

2.2.3 Infinite lists

The lifting of inclusive relations to recursively defined cpo's is defined in terms of finite approximations. It is not obvious what that means for proof obligations in fixpoint induction. In the rest of this chapter we will examine this lifted relation on the example of infinite lists. In this example the lifted relation behaves as expected: this means that two values are related if they have the same "structure" and elements in the structure are pairwise related.

Infinite lists In the last section the cpo of infinite lists were defined as

$$\mathsf{List}\,\mathbf{A} = \prod_{\mathbf{i}} \mathbf{A}$$

for a cpo $\mathbf{A}$. We may also define this cpo using the inverse limit construction as

$$\mathsf{List}\,\mathbf{A} = \mathsf{rec}\,\mathbf{X}.\mathbf{A} \times \mathbf{X}$$

In the rest of the chapter we will examine this recursively define cpo and show how an inclusive relation between values of two cpo's can be lifted to lists of values from these two cpo's.

Construction Let $\mathbf{A}$ be a cpo and define

$$\mathsf{List}\,\mathbf{A} = \mathsf{rec}\,\mathbf{X}.\mathbf{A} \times \mathbf{X}$$

When inspecting the elements in the lists we may use the bijections:

$$\Phi : \mathsf{List}\,\mathbf{A} \to \mathbf{A} \times \mathsf{List}\,\mathbf{A}$$
$$\Psi : \mathbf{A} \times \mathsf{List}\,\mathbf{A} \to \mathsf{List}\,\mathbf{A}$$

Selection On the domain $\mathsf{List}\,\mathbf{A}$ we will define a family of functions $\mathsf{proj}_{\mathbf{i}} : \mathsf{List}\,\mathbf{A} \to \mathbf{A}$:

$$\mathsf{proj}_0(\mathbf{x}) = \perp_{\mathbf{A}}$$
$$\mathsf{proj}_1(\mathbf{x}) = \mathsf{fst}(\Phi(\mathbf{x}))$$
$$\mathsf{proj}_{\mathbf{i}+1}(\mathbf{x}) = \mathsf{proj}_{\mathbf{i}}(\mathsf{snd}(\Phi(\mathbf{x})))$$

The function $\mathsf{proj}_{\mathbf{i}}$ is essentially the same as the function $\mathsf{sel}_{\mathbf{i}}$ defined earlier. It may, however, be confusing that the elements in $\mathsf{List}\,\mathbf{A}$ are defined as infinite lists of finite approximations. The $\mathbf{i}^{\mathrm{th}}$ approximation to a list $\mathbf{x}$ is the list with the first $\mathbf{i}$ elements from $\mathbf{x}$. In the definition we used the function $\theta_{\infty\mathbf{i}}$ to select the $\mathbf{i}^{\mathrm{th}}$ approximation from a list.

Function In the definition of the inverse limit we used a number of auxiliary functions. In this case some of these functions are defined as:

$$\begin{aligned}
&\phi_0(\mathbf{x}) = \langle \bot_{\mathbf{A}}, \bot_{\mathbb{U}} \rangle \\
&\phi_{\mathbf{i}+1}(\mathbf{x}) = \langle \mathsf{fst}(\mathbf{x}), \phi_{\mathbf{i}}(\mathsf{snd}(\mathbf{x})) \rangle \\
&\psi_1(\mathbf{x}) = \bot_{\mathbb{U}} \\
&\psi_{\mathbf{i}+1}(\mathbf{x}) = \langle \mathsf{fst}(\mathbf{x}), \psi_{\mathbf{i}}(\mathsf{snd}(\mathbf{x})) \rangle \\
&\theta_{\mathbf{m}+1\infty}(\mathbf{x})\!\downarrow\!\mathbf{i} = \psi_{\mathbf{i}} \circ \cdots \circ \psi_{\mathbf{m}}(\mathbf{x}), \quad \mathbf{m} \geq \mathbf{i} \\
&\theta'_{\mathbf{m}+1\infty}(\mathbf{x}) = \langle \mathsf{fst}(\mathbf{x}), \theta_{\mathbf{m}\infty}(\mathsf{snd}(\mathbf{x})) \rangle \\
&\Phi(\mathbf{x}) = \bigsqcup_{\mathbf{m}=1}^{\infty} \theta'_{\mathbf{m}\infty}(\theta_{\infty\mathbf{m}}(\mathbf{x})) \\
&\qquad = \langle \mathsf{fst}(\bigsqcup_{\mathbf{m}=1}^{\infty} \theta_{\infty\mathbf{m}}(\mathbf{x})), \bigsqcup_{\mathbf{m}=1}^{\infty} \theta_{\mathbf{m}+1\infty}(\mathsf{snd}(\theta_{\infty\mathbf{m}}(\mathbf{x}))) \rangle \\
&\qquad = \langle \mathsf{fst}(\mathbf{x}\!\downarrow\!1), [\mathsf{snd}(\mathbf{x}\!\downarrow\!1), \mathsf{snd}(\mathbf{x}\!\downarrow\!2), \ldots] \rangle
\end{aligned}$$

Relations Let $\mathbf{A}$ and $\mathbf{B}$ be cpo's related by $\mathbf{R} \in \mathcal{R}_{\mathbf{i}}(\mathbf{A}, \mathbf{B})$. The relations can be lifted to lists as

$$\mathbf{x}\ \mathbf{R}_\infty\ \mathbf{y} \Leftrightarrow \forall\, \mathbf{m}.\mathbf{x}\!\downarrow\!\mathbf{m}\ \mathbf{R}_{\mathbf{m}}\ \mathbf{y}\!\downarrow\!\mathbf{m}$$

where $\mathbf{R}_{\mathbf{m}}$ can be expressed recursively as

$$\begin{aligned}
&\mathbf{x}\ \mathbf{R}_1\ \mathbf{y} \Leftrightarrow \mathsf{fst}(\mathbf{x})\ \mathbf{R}\ \mathsf{fst}(\mathbf{y}) \\
&\mathbf{x}\ \mathbf{R}_{\mathbf{m}+1}\ \mathbf{y} \Leftrightarrow \mathsf{fst}(\mathbf{x})\ \mathbf{R}\ \mathsf{fst}(\mathbf{y}) \wedge \mathsf{snd}(\mathbf{x})\ \mathbf{R}_{\mathbf{m}}\ \mathsf{snd}(\mathbf{y})
\end{aligned}$$

Theorem Let $\mathbf{A}$ and $\mathbf{B}$ be cpo's related by an inclusive relation $\mathbf{R} \in \mathcal{R}_{\mathbf{i}}(\mathbf{A}, \mathbf{B})$. For the recursively defined relation $\mathbf{R}_\infty \in \mathcal{R}_{\mathbf{i}}(\mathsf{List}\,\mathbf{A}, \mathsf{List}\,\mathbf{B})$ we have:

$$\forall\, \mathbf{x} \in \mathsf{List}\,\mathbf{A}, \mathbf{y} \in \mathsf{List}\,\mathbf{B}.\mathbf{x}\ \mathbf{R}_\infty\ \mathbf{y} \Leftrightarrow \forall\, \mathbf{i}.\ \mathsf{proj}_{\mathbf{i}}(\mathbf{x})\ \mathbf{R}\ \mathsf{proj}_{\mathbf{i}}(\mathbf{y})$$

Proof Given $\mathbf{x} \in \mathsf{List}\,\mathbf{A}$ and $\mathbf{y} \in \mathsf{List}\,\mathbf{B}$ the definition says that

$$\mathbf{x}\ \mathbf{R}_\infty\ \mathbf{y} \Leftrightarrow \forall\, \mathbf{i}.\mathbf{x}\!\downarrow\!\mathbf{i}\ \mathbf{R}_{\mathbf{i}}\ \mathbf{y}\!\downarrow\!\mathbf{i}$$

We will prove that for any $\mathbf{i}$

$$\mathbf{x}\!\downarrow\!\mathbf{i}\ \mathbf{R}_{\mathbf{i}}\ \mathbf{y}\!\downarrow\!\mathbf{i} \Leftrightarrow \forall\, \mathbf{j}.\mathbf{i} \leq \mathbf{j} \leq \mathbf{i} \Rightarrow \mathsf{proj}_{\mathbf{j}}(\mathbf{x})\ \mathbf{R}\ \mathsf{proj}_{\mathbf{j}}(\mathbf{y})$$

The proof is by induction. For $\mathbf{i} = 1$ we have

$$\begin{aligned}
\mathbf{x}\!\downarrow\!1\ \mathbf{R}_1\ \mathbf{y}\!\downarrow\!1 &\Leftrightarrow \mathsf{fst}(\mathbf{x}\!\downarrow\!1)\ \mathbf{R}\ \mathsf{fst}(\mathbf{y}\!\downarrow\!1) \\
&\Leftrightarrow \mathsf{proj}_1(\mathbf{x})\ \mathbf{R}\ \mathsf{proj}_1(\mathbf{y})
\end{aligned}$$

Now, assume we have proved for some $\mathbf{i}$ that

$$\mathbf{x}{\downarrow}\mathbf{i}\ \mathbf{R_i}\ \mathbf{y}{\downarrow}\mathbf{i} \Leftrightarrow \forall \mathbf{j}.1 \leq \mathbf{j} \leq \mathbf{i} \Rightarrow \mathsf{proj}_\mathbf{j}(\mathbf{x})\ \mathbf{R}\ \mathsf{proj}_\mathbf{j}(\mathbf{y})$$

then

$$\begin{aligned}
&\mathbf{x}{\downarrow}(\mathbf{i}+1)\ \mathbf{R_{i+1}}\ \mathbf{y}{\downarrow}(\mathbf{i}+1)\\
&\quad\Leftrightarrow \mathsf{fst}(\mathbf{x}{\downarrow}(\mathbf{i}+1))\ \mathbf{R}\ \mathsf{fst}(\mathbf{y}{\downarrow}(\mathbf{i}+1)) \wedge\\
&\qquad \mathsf{snd}(\mathbf{x}{\downarrow}(\mathbf{i}+1))\ \mathbf{R_i}\ \mathsf{snd}(\mathbf{y}{\downarrow}(\mathbf{i}+1))\\
&\quad\Leftrightarrow \mathsf{fst}(\Phi(\mathbf{x}))\ \mathbf{R}\ \mathsf{fst}(\Phi(\mathbf{y})) \wedge\\
&\qquad \mathsf{snd}(\Phi(\mathbf{x})){\downarrow}\mathbf{i}\ \mathbf{R_i}\ \mathsf{snd}(\Phi(\mathbf{y})){\downarrow}\mathbf{i}\\
&\quad\Leftrightarrow \mathsf{proj}_1(\mathbf{x})\ \mathbf{R}\ \mathsf{proj}_1(\mathbf{y}) \wedge\\
&\qquad \forall \mathbf{j}.\mathbf{i} \leq \mathbf{j} \leq \mathbf{i} \Rightarrow \mathsf{proj}_\mathbf{j}(\mathsf{snd}(\Phi(\mathbf{x})))\ \mathbf{R}\ \mathsf{proj}_\mathbf{j}(\mathsf{snd}(\Phi(\mathbf{y})))\\
&\quad\Leftrightarrow \forall \mathbf{j}.\mathbf{i} \leq \mathbf{j} \leq \mathbf{i}+1 \Rightarrow \mathsf{proj}_\mathbf{j}(\mathbf{x})\ \mathbf{R}\ \mathsf{proj}_\mathbf{j}(\mathbf{y})
\end{aligned}$$

□

Comment The theorem shows that when lifting a relation to lists the resulting relation behaves as one would expect: two lists are related if and only if all their elements are related.

2.3 Grammars

A description of the possible objects in a language is the basis for any further analysis of the language. The syntax of a language is often described by a context-free grammar.

Informally a context-free grammar is a set of productions of the form

$$\mathbf{q}_0\ ::= \mathbf{q}_1 \ldots \mathbf{q_n}$$

where the symbol on the left hand side ($\mathbf{q}_0$) is called *nonterminal* and the symbols on the right hand side are either nonterminals, which appear on the left hand side of some of the productions in the grammar, or they are *terminal* symbols. We will often say that it is a production *for* the nonterminal.

One of the nonterminals is called the root symbol. It should be possible to obtain a string of terminal symbols by repeatedly substituting nonterminals with one of the right hand sides of their productions, starting with the root symbol.

A grammar is said to *recognise* the set of finite strings of terminal symbols which can be generated in this way. The set of finite terminal strings is called the *language* defined by the grammar.

The grammar is said to be *unambiguous* if a string of terminal symbols can be generated in at most one way from the root symbol. Among the deterministic grammars, smaller classes of grammars can be characterised as LR(1), LALR(1),

SLR(1), etc. These characterisations will not be discussed any further here but in examples and applications grammars are assumed to be LALR(1). For further discussion of classes of grammars and recognition the reader is referred to [Aho, Sethi & Ullman 1986].

More formally we may define a context-free grammar as follows.

2.3.1 Context-Free Grammar

A *context-free grammar* $\mathbf{G}$ is a quadruple $\mathbf{G} = (\mathbf{N}, \mathbf{T}, \mathbf{Z}, \mathbf{P})$ where

$\mathbf{N}$ is a finite set of *nonterminal* symbols.

$\mathbf{T}$ is a countable set of *terminals* ($\mathbf{T} \cap \mathbf{N} = \varnothing$).

$\mathbf{Z}$ is a *root symbol* in $\mathbf{N}$: $\mathbf{Z} \in \mathbf{N}$.

$\mathbf{P}$ is a countable set of operator symbols ($\mathbf{P} \cap \mathbf{T} = \varnothing$) called *productions* with two functions $\mathsf{lhs} : \mathbf{P} \rightarrow \mathbf{N}$ (called the *left hand side*) and $\mathsf{rhs} : \mathbf{P} \rightarrow (\mathbf{N} \cup \mathbf{T})^*$ (called the *right hand side*). For a production $\mathbf{p} \in \mathbf{P}$ with $\mathsf{lhs}(\mathbf{p}) = \nu$ and $\mathsf{rhs}(\mathbf{p}) = \langle \mathbf{q_1}, \ldots, \mathbf{q_n} \rangle$ we write

$$\mathbf{p} \ : \ \nu \ ::= \mathbf{q_1} \ldots \mathbf{q_n}$$

and we define a mapping $\mathsf{lg} : \mathbf{P} \rightarrow \mathbb{N}$ by $\mathsf{lg}(\mathbf{p}) = \mathbf{n}$ where $\mathbf{n}$ is the length of the right hand side of the production $\mathbf{p}$.

Infinite grammars In the definition we have allowed the sets of terminals and productions to be infinite. In certain approaches to grammars *parameters* are introduced, mainly to describe how names and numbers are supplied by the lexical analyser. From a semantic point of view this is more easily described if we assume that the grammar is *infinite*. Special terminal symbols like `NAME` and `NUMBER` should rather be seen as nonterminals, and the grammar should then be extended with an infinite set of productions of the form:

$$\begin{array}{lll} \texttt{NUMBER} & \rightarrow & 1 \\ & | & 2 \\ & \vdots & \vdots \end{array}$$

Infinite grammars do not cause problems with regard to well-definedness as long as the strings in the language are required to be finite.

2.3.2 Algebraic description

A context-free grammar defines a signature $\Sigma = (\mathbf{N} \cup \mathbf{T}, \mathbf{P} \cup \mathbf{T})$ with

- $\mathbf{N} \cup \mathbf{T}$ as sorts (assuming $\mathbf{N} \cap \mathbf{T} = \varnothing$).
- $\mathbf{P} \cup \mathbf{T}$ as operator symbols (assuming $\mathbf{P} \cap \mathbf{T} = \varnothing$).

The mappings lhs and rhs can be extended to $\mathbf{P} \cup \mathbf{T}$ with lhs as the identity mapping on $\mathbf{T}$ and rhs as the constant mapping with value ϵ on $\mathbf{T}$. An element $\mathbf{x}$ in $\mathbf{P} \cup \mathbf{T}$ will be a terminal iff $\mathsf{rhs}(\mathbf{x}) = \epsilon$.

Σ-Word algebra An algebra over a signature Σ is fully described by carriers (sets) to all sorts and interpretations of all operator symbols such that the interpretation of an operator symbol $\mathbf{p}$ with type $\langle \mathbf{s}_1, \ldots, \mathbf{s_n}, \mathbf{s}_0 \rangle$ belongs to the set $\mathbf{D}_{\mathbf{s}_1} \times \cdots \times \mathbf{D}_{\mathbf{s_n}} \rightarrow \mathbf{D}_{\mathbf{s}_0}$ where $\mathbf{D}_{\mathbf{s_i}}$ is the carrier of sort $\mathbf{s_i}$.

The word algebra (or the Herbrand-universe) for the signature $\Sigma = (\mathbf{N} \cup \mathbf{T}, \mathbf{P} \cup \mathbf{T})$ is defined as follows: Carrier to sort $\mathbf{s} \in \mathbf{N} \cup \mathbf{T}$ is $\mathcal{T}_{\mathbf{s}}$, inductively defined as

$$\mathcal{T}_{\mathbf{t}} = \{\mathbf{t}\}\ , \mathbf{t} \in \mathbf{T}$$
$$\mathcal{T}_{\mathbf{q}} = \{\langle \mathbf{p}, \mathbf{w}_1, \ldots, \mathbf{w_n} \rangle \mid \mathsf{lhs}(\mathbf{p}) = \mathbf{q}, \mathsf{rhs}(\mathbf{p}) = \langle \mathbf{q}_1, \ldots, \mathbf{q_n} \rangle, \mathbf{w_i} \in \mathcal{T}_{\mathbf{s_i}}\}$$

and the set of all words in the algebra is

$$\mathcal{T}_{\Sigma} = \bigcup_{\mathbf{s} \in \mathbf{N} \cup \mathbf{T}} \mathcal{T}_{\mathbf{s}}$$

There are operators $\sigma_{\mathbf{p}}$ for all productions $\mathbf{p} \in \mathbf{P}$:

$$\sigma_{\mathbf{p}}(\mathbf{w}_1, \ldots, \mathbf{w_n}) = \langle \mathbf{p}, \mathbf{w}_1, \ldots, \mathbf{w_n} \rangle$$

where $\mathsf{rhs}(\mathbf{p}) = \langle \mathbf{q}_1, \ldots, \mathbf{q_n} \rangle$ and $\mathbf{w_i} \in \mathcal{T}_{\mathbf{q_i}}$. For all terminal symbols $\mathbf{t} \in \mathbf{T}$ there are 0-ary operators $\sigma_{\mathbf{t}}$:

$$\sigma_{\mathbf{t}} = \mathbf{t}$$

Parse tree For a symbol $\mathbf{s}$ (terminal or nonterminal) we call $\mathcal{T}_{\mathbf{s}}$ the set of *parse trees* with start symbol $\mathbf{s}$. For a grammar $\mathbf{G}$ with root symbol $\mathbf{Z}$ the set of parse trees over the grammar is $\mathcal{T}_{\mathbf{Z}}$. When drawing a parse tree the standard is to label the nodes in the tree with nonterminals. This should only be considered as an abbreviation for the real labels which are the productions. We could not have constructed the word algebra with the same operations if $\mathbf{N} \cup \mathbf{T}$ was the set of operators.

On parse trees with start symbol $\mathbf{q}$ we may define a function

$$\mathsf{flatten}_{\mathbf{q}} : \mathcal{T}_{\mathbf{q}} \rightarrow \mathbf{T}^*$$

by

$$\mathsf{flatten}_{\mathbf{q}}(\mathbf{t}) = \mathbf{t} \quad \text{for } \mathbf{q} = \mathbf{t} \in \mathbf{T}$$
$$\mathsf{flatten}_{\mathbf{q}}(\langle \mathbf{p}, \mathbf{w}_1, \ldots, \mathbf{w}_\mathbf{n} \rangle) = \mathsf{flatten}_{\mathbf{q}_1}(\mathbf{w}_1) \cdots \mathsf{flatten}_{\mathbf{q}_\mathbf{n}}(\mathbf{w}_\mathbf{n})$$

where $\mathsf{rhs}(\mathbf{p}) = \langle \mathbf{q}_1, \ldots \mathbf{q}_\mathbf{n} \rangle$ and $\mathsf{lhs}(\mathbf{p}) = \mathbf{q}$.

2.3.3 Abstract syntax

We have mentioned two different methods to define a language: as a string of terminal symbols generated by the grammar or as parse trees. For unambiguous grammars these definitions are closely related. Each parse tree corresponds to exactly one terminal string as found by the function flatten and it is possible to *parse* a string of terminals (belonging in the language) into a parse tree. If the grammar is LALR(1) this may be done using an automatically generated parser which transforms input into an (often implicit) parse tree.

For ambiguous grammars a string of terminal symbols may correspond to several parse trees. Each of these parse trees can be flattened to the same string of terminal symbols.

Abstract syntax An ambiguous grammar is often simpler and shorter than an unambiguous grammar for the same language. For this reason it may be more attractive to base work on semantics and program analysis on an ambiguous grammar and assume that programs are already represented as parse trees. The phrase *abstract syntax* is often used for this situation. The ambiguous grammar is an *abstract syntax definition* or a *tree grammar* if we assume that the real language is a set of parse trees and not their flattened versions.

The abstract syntax may be further simplified so as not to contain information about some of the terminal symbols and about the order of subtrees.

Concrete syntax The alternative to abstract syntax is to use an unambiguous grammar and let the language be the set of terminal strings which can be derived using the grammar. Such a grammar may be called a *concrete syntax definition*.

Language definition It is common practice to base work on denotational semantics and abstract interpretation on abstract syntax. There can be many reasons for this but the main attraction seems to be that the attention is not distracted by parsing problems like precedence and associativity. Denotational semantics may be used as a tool in language design to discover semantic properties of the language. If the analysis is based on abstract syntax it is possible to construct the semantics description before the precise concrete syntax has been decided. In this view [Schmidt 1986] the abstract syntax is the "real" language and the concrete syntax is derived from the abstract syntax.

The opposite view is that abstract syntax is derived from the concrete syntax by removing all structuring information in the concrete syntax [Gunter, Mosses & Scott 1989]. If this is done in a systematic way we may define a *parser* as a function which maps programs in the concrete syntax into the abstract syntax. The semantics of programs in the concrete syntax can then be given by composing the parser with a semantic functions for programs in the abstract syntax.

Pro et contra The use of abstract syntax makes the specification of a semantics more tractable and it follows a tradition in computer science of solving problems by separation into phases. The gain, however, varies very much according to the size of the grammar. For very small grammars (with only a few productions) there is no real difference between an abstract and a concrete syntax. For "toy languages" of, say 10-20 productions, an abstract syntax may be so much simpler than a concrete syntax that it is possible to grasp the language in a single glance. For real programming languages, however, the gain is more dubious as both a concrete and an abstract syntax is likely to be very complex. The advantage gained from the brevity of the abstract syntax may be cancelled out by the extra effort needed to understand the connection between the abstract and the concrete syntax.

Conclusion The role of abstract syntax in semantics is often to modularise the description and make the semantic definition simpler and free from irrelevant detail. As we shall see in later chapters this is not the only possibility to modularise a specification. The concept of an *attributed scheme* provide a different separation of a specification into two parts. This will be discussed further in chapter 5.

Chapter 3

Frameworks for Abstract Interpretation

In the introduction abstract interpretation was described in general terms as a program analysis method and as a proof technique. In this chapter we will describe and discuss a number of definitions of abstract interpretation. The purpose of abstract interpretation is to answer questions about the behaviour of a program over sets of possible input data without attempting to run the program on the input. To achieve termination of this type of analysis it is therefore important that the question is asked in such a way that "I don't know" is allowed as an answer.

3.1 Fixpoint semantics

The basis for abstract interpretation is formal semantics and to start with, we will consider the role of *fixpoint semantics* and *fixpoint induction* in this type of program analysis. For this purpose we will examine a typical semantics of a simple first order eager functional language as found in [Jones & Mycroft 1986] or [Schmidt 1986]. We will call such a semantics a standard semantics in the sense that it gives the usual meaning to the language. The phrase standard semantics is occasionally used to denote a continuation semantics [Gordon 1979] as opposed to a direct semantics, or to denote a denotational semantics [Stoy 1977] as opposed to a stack or operational semantics.

By *fixpoint semantics* we will here indicate a definition of the semantics of a language as the fixpoint of a continuous function in some cpo. This is essentially the same as denotational semantics or mathematical semantics but we may not want to be constrained by the further structural restrictions and connotation of those formalisms.

Language Consider a language of recursion equations with two syntactic categories. Expressions belong to $\mathcal{E}$ defined as:

$$\begin{array}{lll} \mathbf{exp} ::= & \mathbf{c_i} & \textbf{constants} \\ \mid & \mathbf{x_i} & \textbf{parameters} \\ \mid & \mathbf{a_i(exp_1, \ldots, exp_n)} & \textbf{basic operations} \\ \mid & \mathbf{if\ exp_1\ then\ exp_2\ else\ exp_3} & \textbf{conditional} \\ \mid & \mathbf{f_i(exp_1, \ldots, exp_n)} & \textbf{function calls} \end{array}$$

and programs **prog** belong to the syntactic category $\mathcal{P}$ and have the form:

$$\begin{array}{l} \mathbf{f_1(x_1, \ldots, x_n) = exp_1} \\ \vdots \\ \mathbf{f_k(x_1, \ldots, x_n) = exp_k} \end{array}$$

Without loss of generality we will assume that all functions and basic operations have the same number, **n**, of arguments.

All functions are first order over some set $\mathbf{V}$ and the evaluation is eager. The set $\mathbf{V}$ must contain the boolean values true and false but other constraints on the set and basic operations will not be made here. The syntactic category $\mathcal{E}$ of expressions may contain a countable set of constants $\mathbf{c_i}$ and basic operations $\mathbf{a_i}$. For each constant $\mathbf{c_i}$ there must be a corresponding element $\mathbf{const_i} \in \mathbf{V}$ and for each basic operation $\mathbf{a_i}$ there must be a function $\mathbf{basic_i} : \mathbf{V^n} \to \mathbf{V_\perp}$.

Semantics The semantics will use the following domains

$$\begin{array}{lll} \mathbf{D} & = \mathbf{V_\perp} & \textbf{values} \\ \Theta & = \mathbf{V^n} \to \mathbf{D} & \textbf{functions} \\ \Phi & = \Theta^{\mathbf{k}} & \textbf{function environment} \end{array}$$

There are two semantic functions:

$$\begin{array}{ll} \mathsf{E}[\![\mathbf{exp}]\!] & : \Phi \to \mathbf{V^n} \to \mathbf{D} \\ \mathsf{U}[\![\mathbf{prog}]\!] & : \Phi \end{array}$$

with the definitions

$$\begin{array}{ll} \mathsf{E}[\![\mathbf{c_i}]\!]\phi\nu & = \mathbf{const_i} \\ \mathsf{E}[\![\mathbf{x_i}]\!]\phi\nu & = \nu_{\mathbf{i}} \\ \mathsf{E}[\![\mathbf{a_i(exp_1, \ldots, exp_n)}]\!]\phi\nu & = \mathbf{let\ v_j} = \mathsf{E}[\![\mathbf{exp_j}]\!]\phi\nu, \mathbf{j} = 1, \ldots \mathbf{n\ in} \\ & \quad \mathbf{if\ some\ v_j} = \perp \mathbf{\ then} \perp \mathbf{else} \\ & \quad \mathbf{basic_i(v_1, \ldots, v_n)} \\ \mathsf{E}[\![\mathbf{if\ exp_1\ then\ exp_2\ else\ exp_3}]\!]\phi\nu & = \mathbf{let\ v_1} = \mathsf{E}[\![\mathbf{exp_1}]\!]\phi\nu\ \mathbf{in} \\ & \quad \mathbf{if\ v_1} = \mathsf{true}\ \mathbf{then}\ \mathsf{E}[\![\mathbf{exp_2}]\!]\phi\nu\ \mathbf{else} \end{array}$$

$$\begin{aligned}\mathsf{E}[\![\mathbf{f_i}(\mathbf{exp}_1,\ldots,\mathbf{exp_n})]\!]\phi\nu \quad &= \mathbf{let}\ \mathbf{v_j} = \mathsf{E}[\![\mathbf{exp_j}]\!]\phi\nu, \mathbf{j}=1,\ldots\mathbf{n}\ \mathbf{in}\\ &\quad \mathbf{if\ some}\ \mathbf{v_j} = \bot\ \mathbf{then}\ \bot\ \mathbf{else}\\ &\quad \phi_{\mathbf{i}}(\mathbf{v}_1,\ldots,\mathbf{v_n})\end{aligned}$$

(preceded on the page by the continuation line: $\mathbf{if}\ \mathbf{v}_1 = \mathsf{false}\ \mathbf{then}\ \mathsf{E}[\![\mathbf{exp}_3]\!]\phi\nu\ \mathbf{else}\ \bot$)

and

$$\begin{aligned}\mathsf{U}[\![\ &\mathbf{f}_1(\mathbf{x}_1,\ldots,\mathbf{x_n}) = \mathbf{exp}_1\\ &\vdots\\ &\mathbf{f_k}(\mathbf{x}_1,\ldots,\mathbf{x_n}) = \mathbf{exp_k}]\!] =\\ &\mathsf{fix}\,\lambda\,\phi.\,\langle\mathsf{E}[\![\mathbf{exp}_1]\!]\phi,\ldots,\mathsf{E}[\![\mathbf{exp_k}]\!]\phi\rangle\end{aligned}$$

The semantics is defined as the least fixpoint of a function $\mathbf{F}$ with type $\Phi \to \Phi$ and defined as

$$\mathbf{F}(\phi) = \langle\mathsf{E}[\![\mathbf{exp}_1]\!]\phi,\ldots,\mathsf{E}[\![\mathbf{exp_k}]\!]\phi\rangle$$

The well-definedness of the fixpoint is guaranteed by the continuity of the function $\mathbf{F}$.

Input-output function The meaning of a program is here given as a function environment. We may want to describe the meaning of the program as its *input-output function*. If executing the program means calling the first function in the program with the given input, this can be described as:

$$\begin{aligned}&\mathsf{L}[\![\mathbf{p}]\!] : \mathbf{V^n} \to \mathbf{D}\\ &\mathsf{L}[\![\mathbf{p}]\!] = \mathsf{U}[\![\mathbf{p}]\!]{\downarrow}1\end{aligned}$$

3.1.1 Collecting semantics

The basis for the correctness proof for an abstract interpretation is a *collecting semantics* (or lifted semantics or static semantics). There is some confusion about the terminology but in any definition it will play the role of the most precise (and generally not computable) abstract interpretation. It must from its definition be intuitively correct and the abstract interpretation we want to specify should be proved correct with respect to it. This means that the definition of the collecting semantics restricts the possible abstract interpretations, since abstract interpretations cannot differentiate between behaviours not differentiated in the collecting semantics.

The idea of a collecting semantics as a basis for proofs of a data flow analysis was described by [Cousot & Cousot 1977] under the name of *static semantics*. The phrase *collecting semantics* was coined by [Mycroft 1981]. Some of the ambiguity about this concept may stem from the differences between flow-charts and functional

programs with respect to *state*, *environment*, and *program point*. We will examine four different ways to define a collecting semantics. The main aim is to introduce some of the concepts frequently used in abstract interpretation and program flow analysis. This is done in the simplified context of a first-order eager language.

3.1.2 Meet over all paths

The conceptually easiest way to define a powerset semantics is to use the semantics function $\mathbf{U}$ introduced above.

$$\begin{aligned}
&\mathbf{C}_1[\![\mathbf{prog}]\!] : (\mathbb{P}(\mathbf{V})^{\mathbf{n}} \rightarrow \mathbb{P}(\mathbf{D}))^{\mathbf{k}} \\
&\mathbf{C}_1[\![\mathbf{prog}]\!] = \langle\ \lambda\,\langle \mathbf{S}_1, \ldots, \mathbf{S}_{\mathbf{n}}\rangle\,.\{\mathbf{U}[\![\mathbf{prog}]\!]{\downarrow}1(\mathbf{x}_1, \ldots, \mathbf{x}_{\mathbf{n}})|\mathbf{x}_{\mathbf{j}} \in \mathbf{S}_{\mathbf{j}}\} \\
&\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad \lambda\,\langle \mathbf{S}_1, \ldots, \mathbf{S}_{\mathbf{n}}\rangle\,.\{\mathbf{U}[\![\mathbf{prog}]\!]{\downarrow}\mathbf{k}(\mathbf{x}_1, \ldots, \mathbf{x}_{\mathbf{n}})|\mathbf{x}_{\mathbf{j}} \in \mathbf{S}_{\mathbf{j}}\}\rangle
\end{aligned}$$

The semantic function $\mathbf{C}_1$ is as precise as possible. $\mathbf{U}$ can be extracted from $\mathbf{C}_1$ by calling the resulting functions with singleton sets.

The empty set in $\mathbb{P}(\mathbf{D})$ plays a somewhat trivial role. A function in the program will return the empty set as a result if and only if one of its arguments is the empty set.

This method to define a semantics corresponds to what is known as a *meet over all path* solution to a data flow analysis problem [Hecht 1977]. It is here described as the union of all real program behaviours consistent with the input specification. Notice that the union plays the role of a *meet operation* (eventhough it is in fact a *join operation*). In data flow analysis it is common practice to use domains with an ordering opposite to the ordering of the powerset. This means that the bottom element denotes all possible values and higher values represent more precise information, corresponding to fewer possible behaviours.

The semantic function $\mathbf{C}_1$ is the most precise way to lift a semantics to sets of values. For any possible output described by $\mathbf{C}_1$ there will be input which will actually give this output. The problem with the semantic function $\mathbf{C}_1$ in this context is that it is not directly defined as the fixpoint of a continuous function. This makes it difficult to use in proofs using fixpoint induction.

3.1.3 Independent attribute method

We will now define a semantics with the same functionality as $\mathbf{C}_1$ but now it is defined as a fixpoint. It uses a semantic function $\mathbf{E}_2$ which is $\mathbf{E}_1$ lifted to work on sets.

The phrase *independent attribute method* was introduced by [Jones & Muchnick 1981] in the context of flow-charts. Its meaning in the context of functional

programming is discussed in [Mycroft 1981].

$$
\begin{aligned}
\mathsf{C}_2[\![\mathbf{prog}]\!] &: (\mathbb{P}(\mathbf{V})^{\mathbf{n}} \to \mathbb{P}(\mathbf{D}))^{\mathbf{k}} \\
\mathsf{E}_2[\![\mathbf{exp}]\!] &: (\mathbb{P}(\mathbf{V})^{\mathbf{n}} \to \mathbb{P}(\mathbf{D}))^{\mathbf{k}} \to \mathbb{P}(\mathbf{V})^{\mathbf{n}} \to \mathbb{P}(\mathbf{D})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{E}_2[\![\mathbf{c_i}]\!]\phi\nu &= \{\mathbf{const_i}\} \\
\mathsf{E}_2[\![\mathbf{x_i}]\!]\phi\nu &= \nu_{\mathbf{i}} \\
\mathsf{E}_2[\![\mathbf{a_i}(\mathbf{exp}_1, \ldots, \mathbf{exp_n})]\!]\phi\nu &= \mathbf{let}\ \mathbf{v_j} = \mathsf{E}_2[\![\mathbf{exp_j}]\!]\phi\nu\ \mathbf{in} \\
&\quad \{\mathbf{basic_i}(\mathbf{w}_1, \ldots, \mathbf{w_n}) | \mathbf{w_j} \in \mathbf{v_j} \setminus \{\bot\}\} \\
&\quad \cup(\cup_{\mathbf{j}} \mathbf{v_j} \cap \{\bot\}) \\
\mathsf{E}_2[\![\mathbf{if}\ \mathbf{exp}_1\ \mathbf{then}\ \mathbf{exp}_2\ \mathbf{else}\ \mathbf{exp}_3]\!]\phi\nu &= \\
&\quad \mathbf{let}\ \mathbf{v}_1 = \mathsf{E}_2[\![\mathbf{exp}_1]\!]\phi\nu\ \mathbf{in} \\
&\quad (\mathbf{if}\ \mathsf{true} \in \mathbf{v}_1\ \mathbf{then}\ \mathsf{E}_2[\![\mathbf{exp}_2]\!]\phi\nu\ \mathbf{else}\ \varnothing)\cup \\
&\quad (\mathbf{if}\ \mathsf{false} \in \mathbf{v}_1\ \mathbf{then}\ \mathsf{E}_2[\![\mathbf{exp}_2]\!]\phi\nu\ \mathbf{else}\ \varnothing)\cup \\
&\quad (\mathbf{if}\ \mathbf{v}_1 \setminus \{\mathsf{false}, \mathsf{true}\} = \varnothing\ \mathbf{then}\ \varnothing\ \mathbf{else}\ \{\bot\}) \\
\mathsf{E}_2[\![\mathbf{f_i}(\mathbf{exp}_1, \ldots, \mathbf{exp_n})]\!]\phi\nu &= \mathbf{let}\ \mathbf{v_j} = \mathsf{E}_2[\![\mathbf{exp_j}]\!]\phi\nu\ \mathbf{in} \\
&\quad \phi_{\mathbf{i}}(\mathbf{v}_1 \setminus \{\bot\}, \ldots, \mathbf{v_n} \setminus \{\bot\}) \\
&\quad \cup(\cup_{\mathbf{j}} \mathbf{v_j} \cap \{\bot\})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{C}_2[\![\ \mathbf{f}_1(\mathbf{x}_1, \ldots, \mathbf{x_n}) &= \mathbf{exp}_1 \\
&\vdots \\
\mathbf{f_k}(\mathbf{x}_1, \ldots, \mathbf{x_n}) &= \mathbf{exp_k}]\!] = \\
\mathsf{fix}\,\lambda\,\phi.\,\langle \mathsf{E}_2[\![\mathbf{exp}_1]\!]\phi&, \ldots, \mathsf{E}_2[\![\mathbf{exp_k}]\!]\phi\rangle
\end{aligned}
$$

The fixpoint is well-defined since the functions $\mathsf{E}_2[\![\mathbf{exp}]\!]$ are continuous. It is worth noting that we use a simple powerset $\mathbb{P}(\mathbf{D})$ even though $\mathbf{D}$ is a domain. This means that singleton set function $\{\cdot\} : \mathbf{D} \to \mathbb{P}(\mathbf{D})$ is not a continuous function. We do not, however, need this function and there is no reason here for introducing powerdomains.

The bottom element plays a rather special role in this semantics. Functions are described by maps from sets of non-bottom values to sets of values that may be bottom. Intermediate computations may give bottom values so this must be filtered out before function calls. To describe if one of the arguments to a function may be bottom we use the term $(\cup_{\mathbf{j}} \mathbf{v_j} \cap \{\bot\})$ which is the singleton set $\{\bot\}$ if and only if one of the sets $\mathbf{v_j}$ contain a bottom value. Otherwise it is the empty set.

The semantic function C_2 gives a less precise description than C_1. The result of a function in the program as defined by the C_2 interpretation may include more elements than with the C_1 interpretation. As an example consider the program

$$\mathbf{f}(\mathbf{x}) = \mathbf{x} + \mathbf{x}$$

called with the argument $\{2,3\}$. In the C_1 interpretation the result will be $\{4,6\}$ whereas C_2 will give $\{4,5,6\}$.

The conjecture is that C_2 will give at least the results obtained by C_1.

$$\forall \mathbf{S_i} \subseteq \mathbf{V}, \mathbf{j}: 1,\ldots,\mathbf{k}: (\mathsf{C}_1[\![\mathbf{p}]\!]\!\downarrow\!\mathbf{j})\,\langle \mathbf{S}_1,\ldots \mathbf{S_n}\rangle \subseteq (\mathsf{C}_2[\![\mathbf{p}]\!]\!\downarrow\!\mathbf{j})\,\langle \mathbf{S}_1,\ldots \mathbf{S_n}\rangle$$

This cannot be proved directly by fixpoint induction as C_1 is not defined as a fixpoint, and care must be taken when relating it to U since $(\in)$ is not an inclusive relation.

We can, however, prove that

$$\forall \phi,\nu: \mathsf{E}_2[\![\mathbf{exp}]\!]\phi\nu \supseteq \{\mathsf{E}[\![\mathbf{exp}]\!]\psi\,\mu \,|\, \forall \mathbf{v}: \psi(\mathbf{v}) \in \phi(\{\mathbf{v}\}), \mu_\mathbf{i} \in \nu_\mathbf{i}\}$$

and the result follows from the fact that

$$\mathbf{R} \subseteq (\mathbf{D} \to \mathbf{E}) \times (\mathbb{P}(\mathbf{D}) \to \mathbb{P}(\mathbf{E})):$$
$$\mathbf{f}\,\mathbf{R}\,\mathbf{g} \Leftrightarrow \forall \mathbf{v} \in \mathbf{D}: \mathbf{f}(\mathbf{v}) \in \mathbf{g}(\{\mathbf{v}\})$$

is an inclusive relation when $\mathbf{E}$ has finite height.

3.1.4 Relational method

The collecting semantics C_2 is in some cases less precise than required. It cannot accurately describe that a function will be called with two arguments with the same value and it is difficult to describe that the branches in a conditional expression will only be evaluated with values of parameters which makes the condition true.

For this reason we can define a collecting semantics which can express relationships between parameters. This approach is called the *relational method* [Jones & Muchnick 1981]. Whereas C_2 described input as elements in $(\mathbb{P}(\mathbf{V}))^\mathbf{n}$, that is a list of sets of values, we will here use a set of lists of possible values $\mathbb{P}(\mathbf{V^n})$.

$$\begin{aligned}
&\mathsf{C}_3[\![\mathbf{prog}]\!] &&: (\mathbb{P}(\mathbf{V^n}) \to \mathbb{P}(\mathbf{D}))^\mathbf{k}\\
&\mathsf{E}_3[\![\mathbf{exp}]\!] &&: (\mathbb{P}(\mathbf{V^n}) \to \mathbb{P}(\mathbf{D}))^\mathbf{k} \to \mathbb{P}(\mathbf{V^n}) \to \mathbb{P}(\mathbf{D})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{E}_3[\![\mathbf{c_i}]\!]\phi\nu &&= \{\mathbf{const_i}\}\\
&\mathsf{E}_3[\![\mathbf{x_i}]\!]\phi\nu &&= \{\rho\!\downarrow\!\mathbf{i} | \rho \in \nu\}\\
&\mathsf{E}_3[\![\mathbf{a_i}(\mathbf{exp}_1,\ldots,\mathbf{exp_n})]\!]\phi\nu &&= \mathbf{let}\ \psi_\mathbf{j} = \mathsf{E}_3[\![\mathbf{exp_j}]\!]\phi\ \mathbf{in}\\
& && \quad \cup_{\rho\in\nu}\{\mathbf{basic_i}(\mathbf{w}_1,\ldots,\mathbf{w_n}) | \mathbf{w_j} \in \psi_\mathbf{j}(\{\rho\}) \setminus \{\bot\}\}\\
& && \quad \cup(\cup_\mathbf{j}\psi_\mathbf{j}(\nu) \cap \{\bot\})\\
&\mathsf{E}_3[\![\mathbf{if}\ \mathbf{exp}_1\ \mathbf{then}\ \mathbf{exp}_2\ \mathbf{else}\ \mathbf{exp}_3]\!]\phi\nu = &&\\
& && \quad \mathbf{let}\ \nu_1 = \{\rho \in \nu |\, \mathsf{E}_3[\![\mathbf{exp}_1]\!]\phi\{\rho\} = \mathsf{true}\}\\
& && \quad \phantom{\mathbf{let}}\ \nu_2 = \{\rho \in \nu |\, \mathsf{E}_3[\![\mathbf{exp}_1]\!]\phi\{\rho\} = \mathsf{false}\}\\
& && \quad \phantom{\mathbf{let}}\ \mathbf{v}_1 = \mathsf{E}_3[\![\mathbf{exp}_1]\!]\phi\nu\\
& && \quad \mathbf{in}\ \ \mathsf{E}_3[\![\mathbf{exp}_2]\!]\phi\nu_1 \cup \mathsf{E}_3[\![\mathbf{exp}_3]\!]\phi\nu_2 \cup\\
& && \quad \phantom{\mathbf{in}}\ \ (\mathbf{if}\ \mathbf{v}_1 \setminus \{\mathsf{true},\mathsf{false}\} = \varnothing\ \mathbf{then}\ \varnothing\ \mathbf{else}\ \bot)\\
&\mathsf{E}_3[\![\mathbf{f_i}(\mathbf{exp}_1,\ldots,\mathbf{exp_n})]\!]\phi\nu &&= \mathbf{let}\ \psi_\mathbf{j} = \mathsf{E}_3[\![\mathbf{exp_j}]\!]\phi\ \mathbf{in}\\
& && \quad \cup_{\rho\in\nu}\{\phi_\mathbf{i}(\{\mathbf{v}_1\},\ldots,\{\mathbf{v_n}\}) | \mathbf{v_j} \in \psi_\mathbf{j}(\{\rho\}) \setminus \{\bot\}\}\\
& && \quad \cup(\cup_\mathbf{j}\psi_\mathbf{j}(\nu) \cap \{\bot\})
\end{aligned}$$

$$
\begin{aligned}
\mathsf{C}_3[\![\ & \mathbf{f}_1(\mathbf{x}_1,\ldots,\mathbf{x_n}) = \mathbf{exp}_1 \\
& \qquad\vdots \\
& \mathbf{f_k}(\mathbf{x}_1,\ldots,\mathbf{x_n}) = \mathbf{exp_k}]\!] = \\
& \mathsf{fix}\,\lambda\,\phi.\,\langle \mathsf{E}_3[\![\mathbf{exp}_1]\!]\phi,\ldots,\mathsf{E}_3[\![\mathbf{exp_k}]\!]\phi\rangle
\end{aligned}
$$

Continuing the example with the program

$$\mathbf{f}(\mathbf{x}) = \mathbf{x} + \mathbf{x}$$

the interpretation C_3 will when called with the arguments $\{2,3\}$ return $\{4,6\}$. This information cannot be obtained from the semantics C_2 as the domain $(\mathbb{P}(\mathbf{V})^{\mathbf{n}})$ contains fewer elements than $\mathbb{P}(\mathbf{V^n})$ for non-trivial sets $\mathbf{V}$.

The semantics C_3 is not directly comparable to C_1 and C_2 as they have different functionality.

3.1.5 Minimal function graph

One restriction in the previous definitions of a collecting semantics is that it only collects results from function calls and not arguments to functions in intermediate calls. Instead of representing the function environment as a list of mathematical functions, each function can be represented as an (unordered) list of argument-result pairs. This approach is called the *minimal function graph* [Jones & Mycroft 1986]. This description here is changed in that it uses the more intuitive domain $(\mathbb{P}(\mathbf{V^n} \times \mathbf{D}))^{\mathbf{k}}$ of argument-result pairs to describe the function environment rather than the "double-lifted" domain $(\mathbf{V^n} \to \mathbf{D}_\perp)^{\mathbf{k}}$.

$$
\begin{aligned}
&\mathsf{C}_4[\![\mathbf{prog}]\!] && : \mathbb{P}(\mathbf{V^n}) \to (\mathbb{P}(\mathbf{V^n} \times \mathbf{D}))^{\mathbf{k}} \\
&\mathsf{E}_4[\![\mathbf{exp}]\!] && : (\mathbb{P}(\mathbf{V^n} \times \mathbf{D}))^{\mathbf{k}} \to \mathbb{P}(\mathbf{V^n}) \to (\mathbf{D} \times \mathbb{P}(\mathbf{V^n})^{\mathbf{k}})
\end{aligned}
$$

The semantic function C_4 takes a description of calls to the first function in the program and returns a function environment. The function E_4 takes a parameter and a function environment and returns the value of the expression plus a list of arguments to functions in intermediate calls.

$$
\begin{aligned}
&\mathsf{E}_4[\![\mathbf{c_i}]\!]\phi\nu = \langle \mathbf{const_i}, \mathbf{nocall}\rangle \\
&\mathsf{E}_4[\![\mathbf{x_i}]\!]\phi\nu = \langle \nu_{\mathbf{i}}, \mathbf{nocall}\rangle \\
&\mathsf{E}_4[\![\mathbf{a_i}(\mathbf{exp}_1,\ldots,\mathbf{exp_n})]\!]\phi\nu = \\
&\qquad\qquad \mathbf{let}\ (\mathbf{v_j},\mathbf{c_j}) = \mathsf{E}_4[\![\mathbf{exp_j}]\!]\phi\nu\ \mathbf{in} \\
&\qquad\qquad \mathbf{if\ some}\ \mathbf{v_j} = \perp\ \mathbf{then}\ \langle \perp, \mathbf{c}_1 \sqcup \cdots \sqcup \mathbf{c_n}\rangle \\
&\qquad\qquad \mathbf{else}\ \langle \mathbf{basic_i}(\mathbf{v}_1,\ldots,\mathbf{v_n}), \mathbf{c}_1 \sqcup \cdots \sqcup \mathbf{c_n}\rangle \\
&\mathsf{E}_4[\![\mathbf{if}\ \mathbf{exp}_1\ \mathbf{then}\ \mathbf{exp}_2\ \mathbf{else}\ \mathbf{exp}_3]\!]\phi\nu = \\
&\qquad\qquad \mathbf{let}\ (\mathbf{v}_1,\mathbf{c}_1) = \mathsf{E}_4[\![\mathbf{exp}_1]\!]\phi\nu\ \mathbf{in} \\
&\qquad\qquad \mathbf{if}\ \mathbf{v}_1 = \mathsf{true}\ \mathbf{then\ let}\ (\mathbf{v}_2,\mathbf{c}_2) = \mathsf{E}_4[\![\mathbf{exp}_2]\!]\phi\nu\ \mathbf{in}\ \langle \mathbf{v}_2, \mathbf{c}_1 \sqcup \mathbf{c}_2\rangle\ \mathbf{else}
\end{aligned}
$$

$$
\begin{array}{l}
\quad\quad \textbf{if } \mathbf{v}_1 = \mathsf{false} \textbf{ then let } (\mathbf{v}_3, \mathbf{c}_3) = \mathsf{E}_4[\![\mathbf{exp}_3]\!]\phi\nu \textbf{ in } \langle \mathbf{v}_3, \mathbf{c}_1 \sqcup \mathbf{c}_3 \rangle \\
\quad\quad \textbf{else } \langle \bot, \mathbf{c}_1 \rangle \\
\mathsf{E}_4[\![\mathbf{f_i}(\mathbf{exp}_1, \ldots, \mathbf{exp_n})]\!]\phi\nu = \\
\quad\quad \textbf{let } (\mathbf{v_j}, \mathbf{c_j}) = \mathsf{E}_4[\![\mathbf{exp_j}]\!]\phi\nu \textbf{ in} \\
\quad\quad \textbf{if some } \mathbf{v_j} = \bot \textbf{ then } \langle \bot, \mathbf{c}_1 \sqcup \cdots \sqcup \mathbf{c_n} \rangle \\
\quad\quad \textbf{else if } \langle \mathbf{v}_1, \ldots, \mathbf{v_n}, \mathbf{r} \rangle \in \phi{\downarrow}\mathbf{i} \wedge \mathbf{r} \neq \bot \\
\quad\quad \textbf{then } \langle \mathbf{r}, \mathbf{only_i}(\{\langle \mathbf{v}_1, \ldots, \mathbf{v_n} \rangle\}) \sqcup \mathbf{c}_1 \sqcup \cdots \sqcup \mathbf{c_n} \rangle \\
\quad\quad \textbf{else } \langle \bot, \mathbf{only_i}(\{\langle \mathbf{v}_1, \ldots, \mathbf{v_n} \rangle\}) \sqcup \mathbf{c}_1 \sqcup \cdots \sqcup \mathbf{c_n} \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{C}_4[\![\; \mathbf{f}_1(\mathbf{x}_1, \ldots, \mathbf{x_n}) = \mathbf{exp}_1 \\
\quad\quad\quad\quad \vdots \\
\quad \mathbf{f_k}(\mathbf{x}_1, \ldots, \mathbf{x_n}) = \mathbf{exp_k}]\!] = \\
\quad \lambda\,\mathbf{C}.\,\mathsf{fix}\,\lambda\,\phi.\; \textbf{let } \psi = \phi \sqcup \mathbf{only}_1(\{\langle \mathbf{a}, \bot \rangle \mid \mathbf{a} \in \mathbf{C}\}) \\
\quad\quad\quad \textbf{let } \mathbf{R_i} = \{\langle \mathbf{a}, \mathsf{E}_4[\![\mathbf{exp_i}]\!]\psi\mathbf{a} \rangle \mid \langle \mathbf{a}, \mathbf{r} \rangle \in (\psi{\downarrow}\mathbf{i}\}, \mathbf{i} = 1, \ldots, \mathbf{k} \textbf{ in} \\
\quad\quad\quad \textbf{let } \mathbf{c} = \bigsqcup\{\zeta \mid \langle \mathbf{a}, \langle \mathbf{r}, \zeta \rangle \rangle \in \mathbf{R_i}\} \textbf{ in} \\
\quad\quad\quad \langle \{\langle \mathbf{a}, \bot \rangle \mid \mathbf{a} \in \mathbf{c}_1\}, \ldots, \{\langle \mathbf{a}, \bot \rangle \mid \mathbf{a} \in \mathbf{c_k}\} \rangle \sqcup \\
\quad\quad\quad \langle \{\langle \mathbf{a}, \mathbf{r} \rangle \mid \langle \mathbf{a}, \langle \mathbf{r}, \mathbf{c} \rangle \rangle \in \mathbf{R}_1\}, \ldots, \{\langle \mathbf{a}, \mathbf{r} \rangle \mid \langle \mathbf{a}, \langle \mathbf{r}, \mathbf{c} \rangle \rangle \in \mathbf{R_k}\} \rangle
\end{array}
$$

where $\mathbf{only_i}(\mathbf{v})$ returns a $\mathbf{k}$-tuple where the $\mathbf{i}^{\text{th}}$ element is $\mathbf{v}$ and all other elements are empty sets. The tuple **nocall** is a $\mathbf{k}$-tuple of empty sets. In this semantics the expression evaluation not only produce the result of the expression but it also records all arguments to functions called during evaluation. In the fixpoint iteration, $\mathbf{R_i}$ will contain the results of executing the body of the $\mathbf{i}^{\text{th}}$ function with all arguments in the previous function environment. From this all new function calls and the results are recorded in the new function environment.

Correctness The validity of the minimal function graph interpretation was shown in [Jones & Mycroft 1986] by proving that any result obtained by the minimal function graph could also be obtained in the standard semantics. That is: for all program $\mathbf{p}$ and all input specifications $\mathbf{C} \in \mathbb{P}(\mathbf{V^n})$ we have

$$\forall\, \mathbf{j} = 1..\mathbf{n}, \mathbf{a} \in \mathbf{V^n}, \mathbf{r} \in \mathbf{V}.\, \langle \mathbf{a}, \mathbf{r} \rangle \in (\mathsf{C}_4[\![\mathbf{p}]\!]\mathrm{C}{\downarrow}\mathbf{j} \Rightarrow \mathsf{U}[\![\mathbf{p}]\!]{\downarrow}\mathbf{j}(\mathbf{a})) = \mathbf{r}$$

We may however prove a stronger result: that all real results can be obtained by the minimal function graph. For all programs $\mathbf{p}$:

$$\forall\, \mathbf{a} \in \mathbf{V^n}, \mathbf{r} \in \mathbf{D}.(\mathsf{U}[\![\mathbf{p}]\!]{\downarrow}1)(\mathbf{a}) = \mathbf{r} \Rightarrow \langle \mathbf{a}, \mathbf{r} \rangle \in \mathsf{C}_4[\![\mathbf{p}]\!]\{\mathbf{a}\}{\downarrow}1$$

Proof A proof of the first part can be found in [Jones & Mycroft 1986]. We will here prove the second part. Assume without loss of generality that there is only one function in the program. This means that a function environment will only contain the description of a single function.

Let $\mathbf{p}$ be a program, $\mathbf{a} \in \mathbf{V^n}$ and $\mathbf{r} \in \mathbf{V}$. We will write $\phi_{4,\mathbf{i}}$ for the function environment in the $\mathbf{i}^{\text{th}}$ iteration of $\mathsf{C}_4[\![\mathbf{p}]\!]\{\mathbf{a}\}$ and let $\phi_{0,\mathbf{i}}$ be the function environment in the $\mathbf{i}^{\text{th}}$ iteration of $\mathsf{U}[\![\mathbf{p}]\!]$.

There is nothing to prove if $\mathsf{U}[\![\mathbf{p}]\!](\mathbf{a}) = \bot$ so we assume that there exists an ℓ such that $\phi_{0,\ell}(\mathbf{a}) = \mathbf{r}$. All this tells us is that for any argument which gives a defined result there will be a limit to the recursion depth of the program. The central part in the proof is to see that in the standard semantics the function environment only needs to be defined for the arguments needed in the evaluation:

$$
\begin{aligned}
&\forall\, \phi, \mathbf{v}, \mathbf{s}.\ \mathsf{E}[\![\mathbf{E}]\!]\phi\mathbf{v} = \mathbf{s} \Leftrightarrow \\
&\quad \mathsf{E}_4[\![\mathbf{E}]\!]\{\langle \mathbf{w}, \mathbf{t}\rangle \mid \phi(\mathbf{w}) = \mathbf{t}\}\mathbf{v} = \langle \mathbf{s}, \mathbf{C}\rangle \\
&\quad \mathsf{E}[\![\mathbf{E}]\!](\lambda\, \mathbf{w}.\ \mathbf{if}\ \mathbf{w} \in \mathbf{C}\ \mathbf{then}\ \phi(\mathbf{w})\ \mathbf{else}\ \bot)\mathbf{v} = \mathbf{s}
\end{aligned}
$$

We use this observation to define a sequence of sets

$$
\begin{aligned}
&\mathbf{C}_0 = \{\mathbf{a}\} \\
&\mathbf{C}_{\mathbf{i}+1} = \mathbf{C}_{\mathbf{i}} \cup \bigcup_{\mathbf{v} \in \mathbf{C}_{\mathbf{i}}} \mathsf{snd}(\mathsf{E}_4[\![\mathbf{E}]\!]\{\langle \mathbf{w}, \mathbf{t}\rangle \mid \phi_0(\mathbf{w}) = \mathbf{t}\}\mathbf{v})
\end{aligned}
$$

The limit of this sequence of sets is $\mathbf{C}_\ell$. We now define

$$
\psi = \{\langle \mathbf{w}, \mathbf{t}\rangle \mid \mathbf{w} \in \mathbf{C}_\ell \wedge (\mathbf{t} = \bot \vee \mathbf{t} = \phi_0(\mathbf{w}))\}
$$

We claim that ψ is a fixpoint to the equation that defines C_4. It is not difficult to see that it is a fixpoint but less obvious that it is the least fixpoint. We know, however that $\mathbf{a}$ must occur in the fixpoint and by induction it follows that all elements in $\mathbf{C}_\ell$ must occur in the fixpoint. □

The correctness theorem provides a strong relationship between the minimal function graph semantics and the standard semantics. All results that can be computed by one semantics can also be computed by the other. This is as close as one can get to an equivalence with the different functionality of the semantics.

The minimal function graph semantics, however, is in a sense stronger than the standard semantics in that it will also give information about intermediate function calls performed during an evaluation. In this way it resembles what is sometimes called an *instrumented semantics*.

3.1.6 Instrumented semantics

The computation time of a program is an internal property of a program as it is not expressed directly in the semantics of the language. The standard semantics defined

the meaning of a program as a simple relationship between input and output, and this does not give any information about the intermediate computations.

We will now define a semantics where the computation time is made explicit. This type of semantics is often called an *instrumented semantics*—a phrase coined by Neil Jones years ago even though there do not seem to be any clear references to this. In this semantics the computation time is made explicit as an external property. The semantics will give the meaning of programs as an extended function environment with results as pairs of value and time spent in the computation. The function environment (ϕ) will have type $(\mathbf{V^n} \to (\mathbf{V} \times \mathbb{N})_\perp)^\mathbf{k}$. This semantics and its use is discussed further in [Rosendahl 1989].

The semantics functions for this instrumented semantics will be called $\mathsf{U_t}$ for programs and $\mathsf{E_t}$ for expressions.

$$
\begin{aligned}
&\mathsf{E_t}[\![\mathbf{c_i}]\!]\phi\nu = \langle \mathbf{const_i}, 1\rangle \\
&\mathsf{E_t}[\![\mathbf{x_i}]\!]\phi\nu = \langle \nu{\downarrow}\mathbf{i}, 1\rangle \\
&\mathsf{E_t}[\![\mathbf{a_i}(\mathbf{E}_1,\ldots,\mathbf{E_n})]\!]\phi\nu = \mathbf{let}\ \langle \alpha_\mathbf{j}, \beta_\mathbf{j}\rangle = \mathsf{E_t}[\![\mathbf{E_j}]\!]\phi\nu \\
&\qquad\qquad \mathbf{in}\ \langle \mathbf{basic_i}(\alpha_1,\ldots,\alpha_\mathbf{n}), 1+\beta_1+\cdots+\beta_\mathbf{n}\rangle \\
&\mathsf{E_t}[\![\mathbf{f_i}(\mathbf{E}_1,\ldots,\mathbf{E_n})]\!]\phi\nu = \mathbf{let}\ \langle \alpha_\mathbf{j}, \beta_\mathbf{j}\rangle = \mathsf{E_t}[\![\mathbf{E_j}]\!]\phi\nu, \mathbf{j} = 1,\ldots,\mathbf{n} \\
&\qquad\qquad \mathbf{in\ if\ some}\ \alpha_\mathbf{j} = \perp\ \mathbf{then}\ \langle \perp, \beta_1+\cdots+\beta_\mathbf{n}\rangle \\
&\qquad\qquad \mathbf{else\ let}\ \langle \alpha_0, \beta_0\rangle = \phi_\mathbf{i}(\alpha_1,\ldots,\alpha_\mathbf{n}) \\
&\qquad\qquad \mathbf{in}\ \langle \alpha_0, 1+\beta_0+\beta_1+\cdots+\beta_\mathbf{n}\rangle \\
&\mathsf{E_t}[\![\mathbf{if}\ \mathbf{E}_1\ \mathbf{then}\ \mathbf{E}_2\ \mathbf{else}\ \mathbf{E}_3]\!]\phi\nu = \\
&\qquad\qquad \mathbf{case}\ \mathsf{E_t}[\![\mathbf{E}_1]\!]\phi\nu\ \mathbf{of} \\
&\qquad\qquad\quad \langle \mathsf{true}, \beta_1\rangle \Rightarrow \mathbf{let}\ \langle \alpha_2, \beta_2\rangle = \mathsf{E_t}[\![\mathbf{E}_2]\!]\phi\nu \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \langle \alpha_2, 1+\beta_1+\beta_2\rangle \\
&\qquad\qquad\quad \langle \mathsf{false}, \beta_1\rangle \Rightarrow \mathbf{let}\ \langle \alpha_3, \beta_3\rangle = \mathsf{E_t}[\![\mathbf{E}_3]\!]\phi\nu \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \langle \alpha_3, 1+\beta_1+\beta_3\rangle \\
&\qquad\qquad\quad \langle _\,, \beta_1\rangle \Rightarrow \langle \mathbf{nil}, 1+\beta_1\rangle \\
&\qquad\qquad \mathbf{end}
\end{aligned}
$$

$$
\mathsf{U_t}[\![\langle \mathbf{p}\rangle]\!] : (\mathbf{V^n} \to (\mathbf{V} \times \mathbb{N})_\perp)^\mathbf{k}
$$

$$
\begin{aligned}
\mathsf{U_t}[\![\ &\mathbf{f}_1(\mathbf{x}_1,\ldots,\mathbf{x_n}) = \mathbf{E}_1 \\
&\vdots \\
&\mathbf{f_k}(\mathbf{x}_1,\ldots,\mathbf{x_n}) = \mathbf{E_k}]\!] = \mathsf{fix}(\lambda\,\phi.\,\langle \mathsf{E_t}[\![\mathbf{E}_1]\!]\phi,\ldots,\mathsf{E_t}[\![\mathbf{E_k}]\!]\phi\rangle)
\end{aligned}
$$

It is not possible to prove this semantics correct with respect to the standard semantics. As with the minimal function graph semantics there are aspects of this semantics which are not part of the standard semantics. We may, however, show that

the input-output function described by this instrumented semantics is equivalent to what is given by the standard semantics. That is:

$$\forall\, \mathbf{p}, \mathbf{j} = 1, \ldots, \mathbf{k}, \mathbf{v} \in \mathbf{V^k}.(\mathsf{U}[\![\mathbf{p}]\!]\downarrow\mathbf{j})(\mathbf{v}) = \mathsf{fst}((\mathsf{U_t}[\![\mathbf{p}]\!]\downarrow\mathbf{j})(\mathbf{v}))$$

This does not give any information about the connection between the standard semantics and the second part of the result returned by $\mathsf{U_t}$. An abstract interpretation based on this semantics will therefore use it as a definition of "computation time". If we want to trust the abstract interpretation we must trust that this semantics gives a reliable definition of how much time is spent in computing an expression.

3.1.7 Powerdomains

We mentioned earlier that the singleton set function

$$\{\cdot\} : \mathbf{D} \to \mathbb{P}(\mathbf{D}) \quad , \quad \mathbf{x} \mapsto \{\mathbf{x}\}$$

is not continuous for domains $\mathbf{D}$. This means that we cannot conclude $\{\mathbf{x}\} \subseteq \{\mathbf{y}\}$ from $\mathbf{x} \sqsubseteq \mathbf{y}$. The lack of continuity (or even monotonicity) makes it difficult to relate a standard semantics to a collecting semantics based on powersets. It may also be difficult to relate the collecting semantics to some abstract interpretations which tries to identify presence or absence of undefined result of computations. A solution to these complications is to introduce *power domains*. A power domain $\wp(\mathbf{D})$ for a domain $\mathbf{D}$ is a subset of $\mathbb{P}(\mathbf{D})$ with a continuous injection function $\{\!|\cdot|\!\} : \mathbf{D} \to \wp(\mathbf{D})$.

There are three main ways to define a power domain (see also [Abramsky & Hankin 1987], [Burn, Hankin & Abramsky 1986], and [Mycroft 1981]):

Hoare power domain $(\wp_{\mathbf{H}}(\mathbf{D}), \sqsubseteq_{\mathbf{H}})$

$$\mathbf{S} \in \wp_{\mathbf{H}}(\mathbf{D}) \Leftrightarrow \forall\, \mathbf{x} \in \mathbf{S}, \mathbf{y} \in \mathbf{D}.\mathbf{y} \sqsubseteq \mathbf{x} \Rightarrow \mathbf{y} \in \mathbf{S}$$
$$\mathbf{S}_1 \sqsubseteq_{\mathbf{H}} \mathbf{S}_2 \Leftrightarrow \forall\, \mathbf{x} \in \mathbf{S}_1.\, \exists\, \mathbf{y} \in \mathbf{S}_2.\mathbf{y} \sqsubseteq \mathbf{x}$$

Smyth power domain $(\wp_{\mathbf{S}}(\mathbf{D}), \sqsubseteq_{\mathbf{S}})$

$$\mathbf{S} \in \wp_{\mathbf{S}}(\mathbf{D}) \Leftrightarrow \forall\, \mathbf{x} \in \mathbf{S}, \mathbf{y} \in \mathbf{D}.\mathbf{x} \sqsubseteq \mathbf{y} \Rightarrow \mathbf{y} \in \mathbf{S}$$
$$\mathbf{S}_1 \sqsubseteq_{\mathbf{S}} \mathbf{S}_2 \Leftrightarrow \forall\, \mathbf{x} \in \mathbf{S}_2.\, \exists\, \mathbf{y} \in \mathbf{S}_1.\mathbf{y} \sqsubseteq \mathbf{x}$$

Plotkin power domain $(\wp_{\mathbf{P}}(\mathbf{D}), \sqsubseteq_{\mathbf{P}})$

$$\mathbf{S} \in \wp_{\mathbf{P}}(\mathbf{D}) \Leftrightarrow \forall\, \mathbf{x}, \mathbf{z} \in \mathbf{S}, \mathbf{y} \in \mathbf{D}.\mathbf{x} \sqsubseteq \mathbf{y} \sqsubseteq \mathbf{z} \Rightarrow \mathbf{y} \in \mathbf{S}$$
$$\mathbf{S}_1 \sqsubseteq_{\mathbf{P}} \mathbf{S}_2 \Leftrightarrow \mathbf{S}_1 \sqsubseteq_{\mathbf{H}} \mathbf{S}_2 \wedge \mathbf{S}_1 \sqsubseteq_{\mathbf{S}} \mathbf{S}_2$$

Notice, however, that a collecting semantics based on powersets is well-defined. It is only when we want to prove relations using continuous functions and fixpoint induction that the need for power domains arises.

3.2 Abstraction/Concretisation

The three main ingredients in an abstract interpretation are an abstract semantics, a standard semantics, and a relationship. When an abstract interpretation is designed, it is not always clear in which order these components are decided. It may be natural to start with the standard semantics but there may be several choices depending on type of analysis. Whether one uses an ordinary semantics, a collecting semantics or an instrumented semantics, depends on abstract semantics and the relationship. The aim of this section is to show how the relationship can be expressed and how an abstract semantics can be induced from a standard semantics.

3.2.1 Embedding

A collecting semantics is a function C which maps a program into an element in a set $\mathbf{M}_1$ of meanings. The abstract interpretation can, in a similar fashion, be specified as a function C' which maps programs into a set $\mathbf{M}_2$ of abstract meanings. The traditional way to prove the correctness of abstract interpretations ([Cousot & Cousot 1977]) is to define two functions

$$\alpha : \mathbf{M}_1 \rightarrow \mathbf{M}_2 \qquad \textit{abstraction}$$
$$\gamma : \mathbf{M}_2 \rightarrow \mathbf{M}_1 \qquad \textit{concretisation}$$

and prove some facts about the functions $\mathsf{C}, \mathsf{C}', \alpha$, and γ.

The intuition behind this approach is that the abstract meaning should only be approximate and each abstract meaning should describe a set of possible actual program behaviours. The links between the real and abstract program behaviours are defined by the maps α and γ, where γ applied to an abstract meaning gives a set of possible actual program behaviours and α applied to a set of possible program behaviours gives a safe abstract description. More formally one assumes that the following relations holds between α and γ:

$$\forall \mathbf{x} \in \mathbf{M}_2 \;\; .\alpha(\gamma(\mathbf{x}) = \mathbf{x}$$
$$\forall \mathbf{S} \in \mathbf{M}_1 \;\; .\gamma(\alpha(\mathbf{S})) \sqsupseteq \mathbf{S}$$

where $\mathbf{M}_1$ is (partially) ordered by a relation $\sqsupseteq$. Here $\mathbf{M}_1$ could be the powerset of program behaviours and the ordering be the subset inclusion. Such pairs of functions were called *retraction pairs* in section 2.2. It may also be called a *Galois connection*.

3.2.2 Abstract domain

As an example we consider some sets which are useful in a constant propagation analysis. To start with, we will simplify the domains as much as possible and assume that the actual program behaviours are described by $\mathbf{M}_1 = \mathbb{P}(\mathbf{V})$. We could let C

return the set of possible results of running the program and the abstract meanings will distinguish between zero, one, or many possible results of running the program.

$$\mathbf{M}_2 = \mathbf{V} \cup \{\bot, \top\}$$

We may also write $\mathbf{V}_\bot^\top$ for the domain $\mathbf{V} \cup \{\bot, \top\}$.

The links between $\mathbf{M}_1$ and $\mathbf{M}_2$ can be defined by α and γ:

$$\begin{array}{llll} \alpha(\varnothing) & = \bot & \gamma(\bot) & = \varnothing \\ \alpha(\{\mathbf{e}\}) & = \mathbf{e} & \gamma(\mathbf{e}) & = \{\mathbf{e}\} \\ \alpha(\{\mathbf{e}_1, \mathbf{e}_2, \ldots\}) & = \top & \gamma(\top) & = \mathbf{V} \end{array}$$

It is not difficult to see that these two functions satisfy the relationships stated above. The set $\mathbf{M}_2$ can be made into a cpo with the ordering

$$\bot \sqsubseteq \mathbf{e} \sqsubseteq \top \quad , \forall\, \mathbf{e} \in \mathbf{V}$$

and with this it is easy to see that α and γ are continuous.

3.2.3 Correctness

The desired relationship between the two interpretations C and C' can now be expressed as the equation.

$$\forall\, \mathbf{p}. \mathsf{C}[\![\mathbf{p}]\!] \subseteq \gamma(\mathsf{C}'[\![\mathbf{p}]\!])$$

From this we can say that if C' produces an element in $\mathbf{V}$ for a program $\mathbf{p}$ then all executions of the program $\mathbf{p}$ will return the same value. Notice however that if C' returns $\top$ the program may still return the same value for all input. This is not a surprise since for all interesting programming languages, it is not decidable whether a program returns the same value for all input. The top element $\top$ is in this case a "don't know" value for the analysis.

3.2.4 Fixpoint induction

To prove the relationship between the two interpretations C and C' we may be able to use fixpoint induction. This requires that both interpretations are expressed as fixpoints of continuous functions. Let us assume the definitions

$$\begin{array}{l} \mathsf{C}[\![\mathbf{p}]\!] = \mathsf{fix}(\mathbf{f}) \\ \mathsf{C}'[\![\mathbf{p}]\!] = \mathsf{fix}(\mathbf{f}') \end{array}$$

for some functions

$$\begin{array}{l} \mathbf{f} : \mathbf{M}_1 \to \mathbf{M}_1 \\ \mathbf{f}' : \mathbf{M}_2 \to \mathbf{M}_2 \end{array}$$

The *fixpoint induction* theorem now says that if α and γ satisfy the relations above and $\mathbf{f}$ and $\mathbf{f}'$ satisfy

$$\forall \mathbf{x} \in \mathbf{M}_1 . \mathbf{f}(\mathbf{x}) \subseteq \gamma(\mathbf{f}'(\alpha(\mathbf{x})))$$
$$\forall \mathbf{x} \in \mathbf{M}_2 . \alpha(\mathbf{f}(\gamma(\mathbf{x}))) \sqsubseteq \mathbf{f}'(\mathbf{x})$$

then the fixpoints will be related as

$$\mathsf{fix}(\mathbf{f}) \subseteq \gamma(\mathsf{fix}(\mathbf{f}'))$$

Of course this just means that instead of proving one inequality we now need to prove two. On the other hand this might be easier to do and there are still ways to make the task simpler.

3.2.5 Induced analysis

We have assumed that the interpretation C is defined by the function $\mathbf{f}$ and the domains of meaning $\mathbf{M}_1$ and $\mathbf{M}_2$ are related by functions α and γ. It may then be possible to derive or induce the function $\mathbf{f}'$ such that the correctness property is satisfied.

The first part of the correctness proof dealt with the abstract domain $\mathbf{M}_2$ and the functions α and γ. Provided $\mathbf{M}_1$ is a complete lattice, the function γ can be defined as

$$\gamma(\mathbf{x}) = \bigsqcup \{ \mathbf{S} \subseteq \mathbf{M}_1 | \alpha(\mathbf{S}) \sqsubseteq \mathbf{x} \}$$

and with this definition α and γ will satisfies the conditions and γ will be continuous (see [Mycroft 1981]). In a similar fashion it is possible to define the function α from the function γ.

The second part of the correctness proof used fixpoint induction to prove that $\mathsf{fix}(\mathbf{f}) \subseteq \gamma(\mathsf{fix}(\mathbf{f}'))$. There is, however, a way to construct a function $\mathbf{f}'$ from $\mathbf{f}$ which satisfies the criterion for fixpoint induction. Provided $\mathbf{M}_2$ is a complete lattice, the function $\mathbf{f}'$ can be defined as

$$\mathbf{f}'(\mathbf{x}) = \bigsqcup \{ \alpha(\mathbf{f}(\gamma(\mathbf{y}))) | \mathbf{y} \in \mathbf{M}_2 \wedge \mathbf{y} \sqsubseteq \mathbf{x} \}$$

An analysis defined in this way is often said to be an *induced analysis* or that the analysis $\mathbf{f}'$ is induced from $\mathbf{f}$ by α and γ.

The problem with this approach is that it does not necessarily give an idea of how to implement the analysis, but it is often possible to simplify the definition without changing the function.

Example Let us assume that the set $\mathbf{V}$ contains the natural numbers ($\mathbb{N}$) and that as a basic operation we have addition. From the normal addition in the collecting semantics:

$$\begin{aligned}&\mathsf{add} : \mathbb{P}(\mathbb{N}) \times \mathbb{P}(\mathbb{N}) \to \mathbb{P}(\mathbb{N})\\ &\mathsf{add}(\mathbf{S}_1, \mathbf{S}_2) = \{\mathbf{x}_1 + \mathbf{x}_2 | \mathbf{x}_1 \in \mathbf{S}_1 \wedge \mathbf{x}_2 \in \mathbf{S}_2\}\end{aligned}$$

we would like to define an "abstract" addition of type

$$\mathsf{add}' : \mathbf{M}_2 \times \mathbf{M}_2 \to \mathbf{M}_2$$

and after the ideas of induced analysis we get

$$\begin{aligned}\mathsf{add}'(\mathbf{x}_1, \mathbf{x}_2) &= \bigsqcup\{\alpha(\mathsf{add}(\gamma(\mathbf{y}_1), \gamma(\mathbf{y}_2))) | \mathbf{y}_1, \mathbf{y}_2 \in \mathbf{M}_2 \wedge \mathbf{y}_1 \sqsubseteq \mathbf{x}_1 \wedge \mathbf{y}_2 \sqsubseteq \mathbf{x}_2\}\\ &= \alpha(\mathsf{add}(\gamma(\mathbf{x}_1), \gamma(\mathbf{x}_2)))\end{aligned}$$

Simplifying this definition further we get

$$\begin{aligned}\mathsf{add}'(\mathbf{x}_1, \mathbf{x}_2) &= \bot && \textbf{if } \mathbf{x}_1 = \bot \textbf{ or } \mathbf{x}_2 = \bot\\ &= \top && \textbf{if } \mathbf{x}_1 = \top \textbf{ or } \mathbf{x}_2 = \top\\ &= \mathbf{x}_1 + \mathbf{x}_2 && \textbf{otherwise}\end{aligned}$$

which is just what one expects.

3.2.6 Constant propagation

We have introduced the notion of an induced analysis above with a simplified example.

None of the collecting semantics defined previously just produced a set of possible values as the meaning of the program. The independent attribute collecting semantics C_2 has type

$$\mathsf{C}_2[\![\mathbf{prog}]\!] \quad : (\mathbb{P}(\mathbf{V})^{\mathbf{n}} \to \mathbb{P}(\mathbf{V}_\bot))^{\mathbf{k}}$$

and there are several possibilities if we want to define a *constant propagation* analysis which can be proved correct with respect to C_2. In fact, the abstract domain is one of the major design decisions in the construction of an abstract interpretation.

Quite arbitrarily we will choose the following type for the abstract interpretation,

$$\mathsf{C}_2'[\![\mathbf{prog}]\!] \quad : ((\mathbf{V}^\top_{\bot'})^{\mathbf{n}} \to (\mathbf{V}^\top_{\bot'}))^{\mathbf{k}}$$

where $\bot'$ now will play the role of $\bot$ in $\mathbf{M}_2$ and it is not the same as the $\bot$ in $\mathbf{V}_\bot$. A different choice of type for the abstract meanings is likely to result in a different kind of analysis.

Abstraction The next point is to establish links between the domains of the two interpretations. Let us define the functions

$$\begin{aligned}&\mathbf{A} : (\mathbb{P}(\mathbf{V})^{\mathbf{n}} \to \mathbb{P}(\mathbf{V}_\perp))^{\mathbf{k}} \to ((\mathbf{V}^\top_{\perp'})^{\mathbf{n}} \to (\mathbf{V}^\top_{\perp'}))^{\mathbf{k}}\\ &\mathbf{A}(\langle \mathbf{f}_1, \ldots, \mathbf{f_k}\rangle) = \langle \alpha'(\mathbf{f}_1), \ldots, \alpha'(\mathbf{f_k})\rangle\end{aligned}$$

and

$$\begin{aligned}&\alpha' : (\mathbb{P}(\mathbf{V})^{\mathbf{n}} \to \mathbb{P}(\mathbf{V}_\perp)) \to (\mathbf{V}^\top_{\perp'})^{\mathbf{n}} \to (\mathbf{V}^\top_{\perp'}))\\ &\alpha'(\mathbf{f}) = \lambda \langle \mathbf{x}_1, \ldots, \mathbf{x_n}\rangle . \alpha(\mathbf{f}(\gamma(\mathbf{x}_1), \ldots, \gamma(\mathbf{x_n})) \setminus \{\perp\})\end{aligned}$$

using the functions α and γ from above. Notice in the definition that a $\perp$-value from $\mathbf{f}$ is ignored and a function $\alpha'(\mathbf{f})$ may return a value in $\mathbf{V}$ even if $\mathbf{f}$ may return $\perp$. The constant propagation analysis we are about to define will therefore find values which, if defined, are constant.

Concretisation The concretisation function Γ

$$\Gamma : ((\mathbf{V}^\top_{\perp'})^{\mathbf{n}} \to (\mathbf{V}^\top_{\perp'}))^{\mathbf{k}} \to (\mathbb{P}(\mathbf{V})^{\mathbf{n}} \to \mathbb{P}(\mathbf{V}_\perp))^{\mathbf{k}}$$

can now be defined from $\mathbf{A}$:

$$\begin{aligned}\Gamma(\langle \mathbf{f}'_1, \ldots, \mathbf{f}'_{\mathbf{k}}\rangle) &= \bigsqcup\{\langle \mathbf{f}_1, \ldots, \mathbf{f_n}\rangle \in (\mathbb{P}(\mathbf{V})^{\mathbf{n}} \to \mathbb{P}(\mathbf{V}_\perp))^{\mathbf{k}} \mid \\ &\qquad \mathbf{A}(\langle \mathbf{f}_1, \ldots, \mathbf{f_n}\rangle) \sqsubseteq \langle \mathbf{f}'_1, \ldots, \mathbf{f}'_{\mathbf{k}}\rangle\}\\ &= \langle \gamma'(\mathbf{f}'_1), \ldots, \gamma'(\mathbf{f}'_{\mathbf{k}})\rangle\end{aligned}$$

where

$$\begin{aligned}\gamma'(\mathbf{f}') &= \bigsqcup\{\mathbf{f} \in \mathbb{P}(\mathbf{V})^{\mathbf{n}} \to \mathbb{P}(\mathbf{V}_\perp) \mid \alpha'(\mathbf{f}) \sqsubseteq \mathbf{f}'\}\\ &= \bigsqcup\{\mathbf{f} \mid \lambda \langle \mathbf{x}_1, \ldots, \mathbf{x_n}\rangle . \alpha(\mathbf{f}(\gamma(\mathbf{x}_1), \ldots, \gamma(\mathbf{x_n})) \setminus \{\perp\}) \sqsubseteq \mathbf{f}'\}\\ &= \lambda \langle \mathbf{x}_1, \ldots, \mathbf{x_n}\rangle . \gamma(\mathbf{f}'(\alpha(\mathbf{x}_1), \ldots, \alpha(\mathbf{x_b}))) \cup \{\perp\}\end{aligned}$$

Induced analysis The next step is to use the abstraction and concretisation functions to induce a constant propagation analysis. This is not difficult.

We define the semantic function

$$\mathsf{E}'_2[\![\mathbf{exp}]\!] : (\mathbf{V}^\top_{\perp'})^{\mathbf{n}} \to (\mathbf{V}^\top_{\perp'}))^{\mathbf{k}} \to (\mathbf{V}^\top_{\perp'})^{\mathbf{n}} \to \mathbf{V}^\top_{\perp'}$$

as an induced analysis

$$\mathsf{E}'_2[\![\mathbf{exp}]\!]\phi\nu = \alpha([\mathsf{E}_2[\![\mathbf{exp}]\!]\Gamma(\phi)\,\langle \gamma(\nu_1) \cup \{\perp\}, \ldots, \gamma(\nu_{\mathbf{n}}) \cup \{\perp\}\rangle] \setminus \{\perp\})$$

We would, however, like to simplify this definition so as to make it easier to implement. For this reason we derive the following definitions for each of the possible expressions:

$$\begin{aligned}\mathsf{E}'_2[\![\mathbf{c_i}]\!]\phi\nu &= \alpha(\{\mathbf{const_i}\})\\ &= \mathbf{const_i}\end{aligned}$$

$$\begin{aligned}\mathsf{E}_2'[\![\mathbf{x_i}]\!]\phi\nu &= \alpha(\gamma(\nu_\mathbf{i}))\\ &= \nu_\mathbf{i}\end{aligned}$$

$$\begin{aligned}&\mathsf{E}_2'[\![\mathbf{a_i}(\mathbf{exp}_1,\ldots,\mathbf{exp_n})]\!]\phi\nu =\\ &\quad= \alpha(\mathbf{let}\ \mathbf{v_j} = \mathsf{E}_2[\![\mathbf{exp_j}]\!]\Gamma(\phi)\,\langle\gamma(\nu_1)\cup\{\bot\},\ldots,\gamma(\nu_\mathbf{n})\cup\{\bot\}\rangle)\ \mathbf{in}\\ &\qquad\{\mathbf{basic_i}(\mathbf{w}_1,\ldots,\mathbf{w_n})|\mathbf{w_j}\in\mathbf{v_j}\setminus\{\bot\}\}\\ &\qquad\cup(\cup_\mathbf{j}\mathbf{v_j}\cap\{\bot\})\setminus\{\bot\})\\ &\quad= \alpha(\mathbf{let}\ \mathbf{v_j} = \gamma(\mathsf{E}_2'[\![\mathbf{exp_j}]\!]\phi\nu\ \mathbf{in}\\ &\qquad\{\mathbf{basic_i}(\mathbf{w}_1,\ldots,\mathbf{w_n})|\mathbf{w_j}\in\mathbf{v_j}\setminus\{\bot\}\}\setminus\{\bot\})\\ &\quad= \mathbf{let}\ \mathbf{v_j} = \mathsf{E}_2'[\![\mathbf{exp_j}]\!]\phi\nu\ \mathbf{in}\\ &\qquad\alpha(\{\mathbf{basic_i}(\mathbf{w}_1,\ldots,\mathbf{w_n})|\mathbf{w_j}\in\gamma(\mathbf{v_j})\}\setminus\{\bot\})\\ &\quad= \mathbf{let}\ \mathbf{v_j} = \mathsf{E}_2'[\![\mathbf{exp_j}]\!]\phi\nu\ \mathbf{in}\ \mathbf{basic_i'}(\mathbf{v}_1,\ldots,\mathbf{v_n})\end{aligned}$$

where $\mathbf{basic_i'}$ is the induced basic operation

$$\mathbf{basic_i'}(\mathbf{v}_1,\ldots,\mathbf{v_n}) = \alpha(\{\mathbf{basic_i}(\mathbf{w}_1,\ldots,\mathbf{w_n})|\mathbf{w_j}\in\gamma(\mathbf{v_j})\}\setminus\{\bot\})$$

induced in the same way as add' was constructed from add above.

In a similar way we can induce interpretation of conditionals and function calls:

$$\begin{aligned}&\mathsf{E}_2'[\![\mathbf{if}\ \mathbf{exp}_1\ \mathbf{then}\ \mathbf{exp}_2\ \mathbf{else}\ \mathbf{exp}_3]\!]\phi\nu =\\ &\qquad\mathbf{let}\ \mathbf{v}_1 = \mathsf{E}_2'[\![\mathbf{exp}_1]\!]\phi\nu\ \mathbf{in}\\ &\qquad(\mathbf{if}\ \mathbf{true}\sqsubseteq\mathbf{v}_1\ \mathbf{then}\ \mathsf{E}_2'[\![\mathbf{exp}_2]\!]\phi\nu\ \mathbf{else}\ \bot')\sqcup\\ &\qquad(\mathbf{if}\ \mathbf{false}\sqsubseteq\mathbf{v}_1\ \mathbf{then}\ \mathsf{E}_2'[\![\mathbf{exp}_2]\!]\phi\nu\ \mathbf{else}\ \bot')\\ &\mathsf{E}_2'[\![\mathbf{f_i}(\mathbf{exp}_1,\ldots,\mathbf{exp_n})]\!]\phi\nu =\\ &\qquad\mathbf{let}\ \mathbf{v_j} = \mathsf{E}_2'[\![\mathbf{exp_j}]\!]\phi\nu\ \mathbf{in}\ \phi_\mathbf{i}(\mathbf{v}_1,\ldots,\mathbf{v_n})\end{aligned}$$

3.2.7 Inclusive Relations

The approach presented so far has been based on the fixpoint theorem formulated as:

$$\begin{aligned}&\forall\,\mathbf{x}\in\mathbf{M}_1.\mathbf{f}(\mathbf{x})\subseteq\gamma(\mathbf{f}'(\alpha(\mathbf{x})))\quad\wedge\\ &\forall\,\mathbf{x}\in\mathbf{M}_2.\alpha(\mathbf{f}(\gamma(\mathbf{x})))\sqsubseteq\mathbf{x}\\ \Downarrow&\\ &\mathsf{fix}(\mathbf{f})\subseteq\gamma(\mathsf{fix}(\mathbf{f}'))\end{aligned}$$

There are other useful fixpoint induction theorems, applicable for other types of analysis.

Concretisation There are two useful theorems which only use the concretisation function.

$$\begin{array}{ll} \mathbf{f} & : \mathbf{D} \to \mathbf{D} \\ \mathbf{g} & : \mathbf{E} \to \mathbf{E} \\ \gamma & : \mathbf{E} \to \mathbf{D} \end{array}$$

$$\begin{array}{ll} & \forall \mathbf{x}, \mathbf{y}. \mathbf{x} \sqsubseteq \gamma(\mathbf{y}) \Rightarrow \mathbf{f}(\mathbf{x}) \sqsubseteq \gamma(\mathbf{g}(\mathbf{y})) \\ \Downarrow & \\ & \mathsf{fix}(\mathbf{f}) \sqsubseteq \gamma(\mathsf{fix}(\mathbf{g})) \end{array}$$

and

$$\begin{array}{ll} & \gamma(\bot) = \bot \wedge \\ & \forall \mathbf{x}, \mathbf{y} : \mathbf{x} \sqsupseteq \gamma(\mathbf{y}) \Rightarrow \mathbf{f}(\mathbf{x}) \sqsupseteq \gamma(\mathbf{g}(\mathbf{y})) \\ \Downarrow & \\ & \mathsf{fix}(\mathbf{f}) \sqsupseteq \gamma(\mathsf{fix}(\mathbf{g})) \end{array}$$

Abstractions Similar theorems can be formulated for the abstraction function.

$$\begin{array}{ll} \mathbf{f} & : \mathbf{D} \to \mathbf{D} \\ \mathbf{g} & : \mathbf{E} \to \mathbf{E} \\ \alpha & : \mathbf{D} \to \mathbf{E} \end{array}$$

$$\begin{array}{ll} & \alpha(\bot) = \bot \wedge \\ & \forall \mathbf{x}, \mathbf{y}. \alpha(\mathbf{x}) \sqsubseteq \mathbf{y} \Rightarrow \alpha(\mathbf{f}(\mathbf{x})) \sqsubseteq \mathbf{g}(\mathbf{y}) \\ \Downarrow & \\ & \alpha(\mathsf{fix}(\mathbf{f})) \sqsubseteq \mathsf{fix}(\mathbf{g}) \end{array}$$

Example: $\mathbf{F}^\sharp$ from strictness analysis [Mycroft 1980].

$$\begin{array}{ll} & \forall \mathbf{x}, \mathbf{y}. \alpha(\mathbf{x}) \sqsupseteq \mathbf{y} \Rightarrow \alpha(\mathbf{f}(\mathbf{x})) \sqsupseteq \mathbf{g}(\mathbf{y}) \\ \Downarrow & \\ & \alpha(\mathsf{fix}(\mathbf{f})) \sqsupseteq \mathsf{fix}(\mathbf{g}) \end{array}$$

Example: $\mathbf{F}^\flat$ from strictness analysis.

Inclusive relations The more general fixpoint induction theorem can be expressed using inclusive relations (see also section 2.2):

For cpos $\mathbf{D}$ and $\mathbf{E}$, let $\mathbf{R}$ be a strict and inclusive relation $\mathbf{R} \subseteq \mathbf{D} \times \mathbf{E}$. For continuous functions $\mathbf{f} : \mathbf{D} \to \mathbf{D}$ and $\mathbf{g} : \mathbf{E} \to \mathbf{E}$, we have

$$\begin{array}{ll} & \forall \mathbf{x} \in \mathbf{D}, \mathbf{y} \in \mathbf{E}. \mathbf{x} \mathbf{R} \mathbf{y} \Rightarrow \mathbf{f}(\mathbf{x}) \mathbf{R} \mathbf{g}(\mathbf{y}) \\ \Downarrow & \\ & \mathsf{fix}(\mathbf{f}) \mathbf{R} \mathsf{fix}(\mathbf{g}) \end{array}$$

This also explains why there are extra conditions about strictness of α and γ in some of the previous theorems.

Examples The following relations are inclusive for continuous functions γ, α, ξ:

$$\mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow \mathbf{x} \sqsubseteq \gamma(\mathbf{y})$$
$$\mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow \mathbf{x} \sqsupseteq \gamma(\mathbf{y})$$
$$\mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow \alpha(\mathbf{x}) \sqsubseteq \mathbf{y}$$
$$\mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow \alpha(\mathbf{x}) \sqsupseteq \mathbf{y}$$
$$\mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow \xi_1(\mathbf{x}, \mathbf{y}) \sqsubseteq \xi_2(\mathbf{x}, \mathbf{y})$$
$$\mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow \forall \mathbf{z} : \xi_1(\mathbf{x}, \mathbf{y}, \mathbf{z}) \sqsubseteq \xi_2(\mathbf{x}, \mathbf{y}, \mathbf{z})$$

but not all relations are inclusive. The following two relations are not inclusive for all continuous functions γ.

$$\mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow \mathbf{x} \in \gamma(\mathbf{y})$$
$$\mathbf{x}\ \mathbf{R}\ \mathbf{y} \Leftrightarrow \mathbf{x} \neq \gamma(\mathbf{y})$$

The use of inclusive relations in abstract interpretation will be considered further in the next section.

3.3 Relational abstract interpretation

In this section we will summarise the approach of [Reynolds 1983] adapted to abstract interpretations.

3.3.1 Language

The language is essentially the same as used by [Reynolds 1983]. The notation and names for certain sets have been adjusted for consistency.

The expressions used in the semantic framework are an extension to the explicitly typed lambda calculus. There are two syntactic categories.

Type expressions belong to the syntactic category $\mathcal{T}$ defined recursively as

$\tau ::= \kappa$	**constant type**
$\mid \quad \upsilon$	**type variable**
$\mid \quad \tau_1 \rightarrow \tau_2$	**function domain**
$\mid \quad \tau_1 \times \tau_2$	**cartesian product**

Type-constants are names belonging to a countable set **C** and type-variables belong to the set **S**. The set **C** will contain the type <u>**bool**</u> of boolean values. The subset of types that does not contain any type variables is called $\mathcal{T}_{\mathbf{c}}$ of fixed types.

Expressions belong to the syntactic category $\mathcal{E}$

$\mathbf{e} ::= $	$\mathbf{c}$	**constant**
$\mid$	$\mathbf{x}$	**parameters**
$\mid$	$\mathbf{s}$	**operator names**
$\mid$	$\mathbf{e}_1(\mathbf{e}_2)$	**application**
$\mid$	$\lambda\,\mathbf{x} : \tau.\mathbf{e}$	**abstraction**
$\mid$	$\langle \mathbf{e}_1, \mathbf{e}_2 \rangle$	**pairing**
$\mid$	$\mathbf{e}.1$	**selection**
$\mid$	$\mathbf{e}.2$	**selection**
$\mid$	$\textbf{if } \mathbf{e}_1 \textbf{ then } \mathbf{e}_2 \textbf{ else } \mathbf{e}_3$	**conditional**
$\mid$	$\textbf{fix } \mathbf{e}$	**fixpoint**

The constants $\mathbf{c}$ are taken from a countable set $\mathbf{K}$ and similarly for parameters $\mathbf{x} \in \mathbf{Id}$, and operator names $\mathbf{s} \in \Omega$.

3.3.2 Type assignment

We are only interested in well-typed expressions and assume that a *type assignment* function maps expressions to type expressions. The type assignment function requires that constants and operator names are typed and other expressions can then be typed inductively. Let there be given two functions,

$$\Upsilon_0 : \mathbf{K} \rightarrow \mathcal{T}_{\mathbf{c}}$$
$$\Upsilon_1 : \Omega \rightarrow \mathcal{T}$$

such that the type of constant $\mathbf{c}$ is $\Upsilon_0(\mathbf{c})$ and the type of operator name $\mathbf{s}$ is $\Upsilon_1(\mathbf{s})$. The type assignment function will have functionality

$$\Upsilon : \mathcal{E} \rightarrow (\mathbf{Id} \rightarrow \mathcal{T}) \rightarrow \mathcal{T}_{\perp}$$

It takes an expression and an environment of parameter types (π) and returns the type. The type assignment is here defined as a function. We could also have used typing rules which is the common practice in language descriptions. Its definition as a function, however, is useful in the rest of the description.

$$\begin{aligned}
\Upsilon[\![\mathbf{c}]\!]\pi &= \Upsilon_0(\mathbf{c}) \\
\Upsilon[\![\mathbf{x}]\!]\pi &= \pi(\mathbf{x}) \\
\Upsilon[\![\mathbf{s}]\!]\pi &= \Upsilon_1(\mathbf{s}) \\
\Upsilon[\![\mathbf{e}_1(\mathbf{e}_2)]\!]\pi &= \textbf{let } \tau_1 \rightarrow \tau_2 = \Upsilon[\![\mathbf{e}_1]\!]\pi \\
&\quad\ \textbf{and } \tau_1 = \Upsilon[\![\mathbf{e}_2]\!]\pi \\
&\quad\quad \textbf{in } \tau_2 \\
\Upsilon[\![\lambda\,\mathbf{x} : \tau.\mathbf{e}]\!]\pi &= \textbf{let } \tau_1 = \Upsilon[\![\mathbf{e}]\!]\pi[\mathbf{x} \mapsto \tau]
\end{aligned}$$

$$
\begin{array}{ll}
& \quad\;\; \textbf{in } \tau \to \tau_1 \\
\Upsilon[\![\langle \mathbf{e}_1, \mathbf{e}_2 \rangle]\!]\pi & = \textbf{let } \tau_1 = \Upsilon[\![\mathbf{e}_1]\!]\pi \\
& \quad \textbf{and } \tau_2 = \Upsilon[\![\mathbf{e}_2]\!]\pi \\
& \quad\;\; \textbf{in } \tau_1 \times \tau_2 \\
\Upsilon[\![\mathbf{e}.1]\!]\pi & = \textbf{let } \tau_1 \times \tau_2 = \Upsilon[\![\mathbf{e}]\!]\pi \textbf{ in } \tau_1 \\
\Upsilon[\![\mathbf{e}.2]\!]\pi & = \textbf{let } \tau_1 \times \tau_2 = \Upsilon[\![\mathbf{e}]\!]\pi \textbf{ in } \tau_2 \\
\Upsilon[\![\textbf{if } \mathbf{e}_1 \textbf{ then } \mathbf{e}_2 \textbf{ else } \mathbf{e}_3]\!]\pi & \\
& = \textbf{let } \underline{\textbf{bool}} = \Upsilon[\![\mathbf{e}_1]\!]\pi \\
& \quad \textbf{and } \tau_2 = \Upsilon[\![\mathbf{e}_2]\!]\pi \\
& \quad \textbf{and } \tau_2 = \Upsilon[\![\mathbf{e}_3]\!]\pi \\
& \quad\;\; \textbf{in } \tau_2 \\
\Upsilon[\![\textbf{fix } \mathbf{e}]\!]\pi & = \textbf{let } \tau_1 \to \tau_1 = \Upsilon[\![\mathbf{e}]\!]\pi \textbf{ in } \tau_1
\end{array}
$$

The phrase $\pi[\mathbf{x} \mapsto \tau]$ is a shorthand for $\lambda\, \mathbf{y}.\ \textbf{if } \mathbf{y} = \mathbf{x} \textbf{ then } \tau \textbf{ else } \pi(\mathbf{y})$

3.3.3 Domain assignment

The expressions in the language will be interpreted as elements in a domain. The interpretation consists of two parts: interpretation of type expressions and interpretation of ordinary expressions. The second part is the meaning assignment which will be described later. The first part is the *domain assignment* which assigns domains to type expressions. The domain assignment is determined by the assignments made to constant types and type variables. From this all other type expressions are assigned types inductively.

The domain assignment is derived from two mappings

$$
\begin{array}{l}
\mathsf{S}_0 : \mathbf{C} \to \mathcal{CPO} \\
\mathsf{S}_1 : \mathbf{S} \to \mathcal{CPO}
\end{array}
$$

where the type constant $\kappa \in \mathbf{C}$ is assigned the domain $\mathsf{S}_0(\kappa)$ and the type variable $\upsilon \in \mathbf{S}$ is assigned the domain $\mathsf{S}_1(\upsilon)$. For $\underline{\textbf{bool}} \in \mathbf{C}$ the domain assignment $\mathsf{S}_0(\underline{\textbf{bool}})$ must give the domain of boolean values $\mathbb{B}_\perp$. The domain assignment function can then be defined as

$$
\mathsf{S} : \mathcal{T} \to \mathcal{CPO}
$$

$$
\begin{array}{ll}
\mathsf{S}[\![\kappa]\!] & = \mathsf{S}_0(\kappa) \\
\mathsf{S}[\![\upsilon]\!] & = \mathsf{S}_1(\upsilon) \\
\mathsf{S}[\![\tau_1 \to \tau_2]\!] & = \mathsf{S}[\![\tau_1]\!] \to \mathsf{S}[\![\tau_2]\!] \\
\mathsf{S}[\![\tau_1 \times \tau_2]\!] & = \mathsf{S}[\![\tau_1]\!] \times \mathsf{S}[\![\tau_2]\!]
\end{array}
$$

where "$\to$" and "$\times$" on the right hand side denote the domain of continuous functions and cartesian product respectively.

3.3.4 Meaning assignment

The ordinary expressions will be assigned meanings as objects of the domains assigned to the type of expressions. The assignment requires that constants and operator symbols are given meanings, and meanings of other expressions are defined inductively. Let there be two given functions E_0 and E_1 such that

$$\forall\, \mathbf{c} \in \mathbf{C}.\, \mathsf{E}_0(\mathbf{c}) \in \mathsf{S}[\![\Upsilon_0(\mathbf{c})]\!]$$
$$\forall\, \mathbf{s} \in \Omega.\, \mathsf{E}_1(\mathbf{s}) \in \mathsf{S}[\![\Upsilon_1(\mathbf{s})]\!]$$

The constant $\mathbf{c}$ of type $\tau_1 = \Upsilon_0(\mathbf{c})$ is assigned the meaning $\mathsf{E}_0(\mathbf{c}) \in \mathsf{S}[\![\tau_1]\!]$ and the operator symbol $\mathbf{s}$ of type τ_2 is assigned the meaning $\mathsf{E}_1(\mathbf{s}) \in \mathsf{S}[\![\tau_2]\!]$

The *meaning assignment* will have functionality

$$\mathsf{E} : \mathcal{E} \to (\mathbf{Id} \to \mathcal{D}) \to \mathcal{D}$$

where

$$\mathcal{D} = \sum_{\tau \in \mathcal{T}} (\mathsf{S}[\![\tau]\!])$$

It takes an expression and an environment of parameter meanings and returns the meaning. The meaning assignment is defined as follows

$$\begin{aligned}
\mathsf{E}[\![\mathbf{c}]\!]\eta &= \mathsf{E}_0(\mathbf{c}) \\
\mathsf{E}[\![\mathbf{v}]\!]\eta &= \eta(\mathbf{v}) \\
\mathsf{E}[\![\mathbf{s}]\!]\eta &= \mathsf{E}_1(\mathbf{s}) \\
\mathsf{E}[\![\mathbf{e}_1(\mathbf{e}_2)]\!]\eta &= \mathsf{E}[\![\mathbf{e}_1]\!]\eta\,(\mathsf{E}[\![\mathbf{e}_2]\!]\eta) \\
\mathsf{E}[\![\lambda\,\mathbf{v} : \tau.\mathbf{e}]\!]\eta &= \lambda\,\mathbf{x}.\, \mathsf{E}[\![\mathbf{e}]\!]\eta[\mathbf{v} \to \mathbf{x}] \\
\mathsf{E}[\![\langle \mathbf{e}_1, \mathbf{e}_2 \rangle]\!]\eta &= \langle \mathsf{E}[\![\mathbf{e}_1]\!]\eta, \mathsf{E}[\![\mathbf{e}_2]\!]\eta \rangle \\
\mathsf{E}[\![\mathbf{e}.1]\!]\eta &= (\mathsf{E}[\![\mathbf{e}]\!]\eta)\downarrow 1 \\
\mathsf{E}[\![\mathbf{e}.2]\!]\eta &= (\mathsf{E}[\![\mathbf{e}]\!]\eta)\downarrow 2 \\
\mathsf{E}[\![\mathbf{if}\ \mathbf{e}_1\ \mathbf{then}\ \mathbf{e}_2\ \mathbf{else}\ \mathbf{e}_3]\!]\eta & \\
&= \mathbf{case}\ \mathsf{E}[\![\mathbf{e}_1]\!]\eta\ \mathbf{of} \\
&\qquad \mathsf{true} \ \Rightarrow \mathsf{E}[\![\mathbf{e}_2]\!]\eta \\
&\qquad \mathsf{false} \ \Rightarrow \mathsf{E}[\![\mathbf{e}_3]\!]\eta \\
&\qquad _ \ \Rightarrow \bot \\
&\quad \mathbf{end} \\
\mathsf{E}[\![\mathbf{fix}\ \mathbf{e}]\!]\eta &= \mathsf{fix}_\tau\,(\mathsf{E}[\![\mathbf{e}]\!]\eta) \\
&\quad \text{where } \mathbf{e} \text{ is assigned the type} \\
&\quad \tau \to \tau \text{ by the type assignment}
\end{aligned}$$

3.3.5 Versions and interpretations

The set of constant types $\mathbf{K}$ and the set of constants $\mathbf{C}$ are supposed to be fixed. They cannot vary between interpretations and their interpretation will always be the same. This means that the functions Υ_0, S_0, and E_0 are fixed in the model.

A *version* of the language will fix the set of type variables and the set of operator names. A version is characterised by the definition of the function Υ_1 and this makes it possible to define the function Υ. In a version the types of expressions are fixed but the interpretation of type variables may vary.

An *interpretation* will assign meanings to expressions. It requires that type variables are assigned domains and that operator symbols are given meanings. An interpretation is characterised by the definition of S_1 and E_1. This makes it possible to define E and S.

3.3.6 Relation Assignment

A version of the language is characterised by the functions $(\Upsilon, \mathsf{S}_0, \mathsf{E}_0)$. We will now consider two interpretations of the same version of the language. The two interpretations will be characterised by the functions $(\mathsf{S}_1', \mathsf{E}_1')$ and $(\mathsf{S}_1'', \mathsf{E}_1'')$ respectively and for the interpretations we can define $(\mathsf{S}', \mathsf{E}')$ and $(\mathsf{S}'', \mathsf{E}'')$.

A *relation assignment* R_1 between interpretations $(\mathsf{S}_1', \mathsf{E}_1')$ and $(\mathsf{S}_1'', \mathsf{E}_1'')$ defines a relation between the interpretations of the type variables.

$$\forall\, \upsilon \in \mathbf{S} : \mathsf{R}_1(\upsilon) \in \mathcal{R}_\mathbf{i}\,(\mathsf{S}_1'(\upsilon), \mathsf{S}_1''(\upsilon))$$

The relations $\mathcal{R}_\mathbf{i}(\mathbf{D}_1, \mathbf{D}_2)$ are the strict and inclusive relations between $\mathbf{D}_1$ and $\mathbf{D}_2$. They are the relations which preserve upper bounds and relate the bottom elements in the domains.

The relation can be extended to all types inductively.

$$\begin{aligned}
\mathsf{R}[\![\kappa]\!] &= \Delta_{\mathsf{S}_0(\kappa)} \\
\mathsf{R}[\![\upsilon]\!] &= \mathsf{R}_1(\upsilon) \\
\mathsf{R}[\![\tau_1 \to \tau_2]\!] &= \mathsf{R}[\![\tau_1]\!] \boxed{\to} \mathsf{R}[\![\tau_2]\!] \\
\mathsf{R}[\![\tau_1 \times \tau_2]\!] &= \mathsf{R}[\![\tau_1]\!] \boxtimes \mathsf{R}[\![\tau_2]\!]
\end{aligned}$$

$\Delta_{\mathsf{S}_0(\kappa)}$ denotes the identity relation and $\boxed{\to}$ and $\boxtimes$ on the right hand side denote the extension of relations for functions spaces and cartesian products. Relationships will be written infix so that $\mathbf{x}\;\mathbf{R}\;\mathbf{y} \Leftrightarrow \langle \mathbf{x}, \mathbf{y} \rangle \in \mathbf{R}$.

The relation can further be extended to environments. Let π be a parameter type environment. A parameter meaning environment of type π is a function η such that $\eta(\mathbf{v}) \in \mathsf{S}[\![\pi(\mathbf{v})]\!]$. Let η' and η'' be parameter meaning environments of type π under interpretations S' and S'' respectively. The environments are related iff

$$\forall\, \mathbf{v} \in \mathbf{P} : \eta'(\mathbf{v})\; \mathsf{R}[\![\pi(\mathbf{v})]\!]\; \eta''(\mathbf{v})$$

and we write $\eta'\; \mathsf{R}^\pi\; \eta''$.

3.3.7 Abstraction Theorem

The abstraction theorem says that interpretations of expressions with related environments give related results. More formally this means,

$$\begin{array}{l} \quad \eta' \ \mathsf{R}^{\pi} \ \eta'' \\ \Downarrow \\ \quad \mathsf{E}'[\![\mathbf{e}]\!]\eta' \ \mathsf{R}[\![\Upsilon[\![\mathbf{e}]\!]\pi]\!] \ \mathsf{E}''[\![\mathbf{e}]\!]\eta'' \end{array}$$

The proof is by structural induction on the syntax of expressions $\mathbf{e}$.

3.3.8 A model for polymorphism

In [Reynolds 1983] this model is used to describe (parametric) polymorphism. The abstraction theorem can be used to prove that if two different implementations of a data type are equivalent in some sense, then uses of the data type in expressions will give equivalent results.

3.4 Automatic complexity analysis

In section 3.1 we defined an instrumented semantics which made "the time spent in computations" an explicit value in the semantics. We may use that semantics as a basis for an abstract interpretation which performs an automatic complexity analysis: We will define a semantics which from the size of input to a program computes an upper bound to the computation time for all input with the given size.

Size of input First of all, we need to give a notion of "size" of input to programs. We will now assume that the programming language uses Lisp's S-expressions as value domain $\mathbf{V}$ and the "size" of input will be a description of the structure of input. We could also describe size by, say, the length of lists, number of cells in a structure, or dimension of matrices. A description of the structure is used for simplicity and does not exclude other size-measures.

The structure of an S-expression will be described by a so-called *partially known structure*. A partially known structure represents a subset of possible S-expressions and can as such be seen as an abstraction (or quotient) of the powerset of S-expressions. The set of partially known structures will be called $\tilde{\mathbf{V}}$ and we introduce so-called tilde-extensions to the basic operations.

The domain of partially known structures A partially known structure is an S-expression containing a special symbol to denote that the given substructure is unknown and therefore could be substituted for any S-expression. The atom all will be used as this special symbol and the set of partially known structures $\tilde{\mathbf{V}}$ can be seen as an abstraction of the powerset of S-expressions ($\mathbb{P}(\mathbf{V})$).

$$\tilde{\mathbf{V}}_\perp \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} \supseteq \mathbb{P}(\mathbf{V})$$

with

$$\begin{array}{lll}\gamma(\mathbf{x}) = & \varnothing & \textbf{if } \mathbf{x} = \perp\ , \textbf{else} \\ & \mathbf{V} & \textbf{if } \mathbf{x} = \mathsf{all}\ , \textbf{else} \\ & \{\mathbf{x}\} & \textbf{if } \mathbf{x} \in \textbf{Atoms}\ , \textbf{else} \\ & \{\langle \mathbf{a}, \mathbf{b}\rangle \mid \mathbf{a} \in \gamma(\mathbf{x}_1) \wedge \mathbf{b} \in \gamma(\mathbf{x}_2)\} & \textbf{if } \mathbf{x} = \langle \mathbf{x}_1, \mathbf{x}_2 \rangle \\ \alpha(\mathbf{x}) = & \perp & \textbf{if } \mathbf{x} = \varnothing\ , \textbf{else} \\ & \mathbf{e} & \textbf{if } \mathbf{x} = \{\mathbf{e}\}\ , \textbf{else} \\ & \mathsf{all} & \textbf{if } \mathbf{x} \cap \textbf{Atoms} \neq \varnothing\ , \textbf{else} \\ & \langle \alpha(\{\mathbf{a} \mid \langle \mathbf{a}, \mathbf{b}\rangle \in \mathbf{x}\}), \alpha(\{\mathbf{b} \mid \langle \mathbf{a}, \mathbf{b}\rangle \in \mathbf{x}\}) \rangle & \end{array}$$

where **Atoms** is the set of atoms in S-expressions. It may be defined as $\textbf{Atoms} = \mathbb{N} + \mathbb{A} + \mathbb{B}$. The set $\tilde{\mathbf{V}}$ may be seen as the S-expressions built from $\widetilde{\textbf{Atoms}} = \mathbb{N} + \mathbb{A} + \mathbb{B} + \{\mathsf{all}\}$ as a separated sum. α and γ can be proved to satisfy the condition for abstraction and concretisation functions [Cousot & Cousot 1977],

$$\begin{array}{ll}\forall \mathbf{S} \in \mathbb{P}(\mathbf{V}). & \gamma(\alpha(\mathbf{S})) \supseteq \mathbf{S} \\ \forall \mathbf{x} \in \tilde{\mathbf{V}}. & \alpha(\gamma(\mathbf{x})) = \mathbf{x}\end{array}$$

The set $\tilde{\mathbf{V}}_\perp$ can be equipped with an ordering $\sqsubseteq$ and a least upper bound **Lub**.

$$\begin{array}{ll}\forall \mathbf{x}, \mathbf{y} \in \tilde{\mathbf{V}}_\perp & .\mathbf{x} \sqsubseteq \mathbf{y} \overset{\text{def}}{\Leftrightarrow} \gamma(\mathbf{x}) \subseteq \gamma(\mathbf{y}) \\ \forall \mathbf{S} \subseteq \tilde{\mathbf{V}}_\perp & .\mathbf{Lub}(\mathbf{S}) \overset{\text{def}}{=} \alpha\left(\bigcup \{\gamma(\mathbf{y}) \mid \mathbf{y} \in \mathbf{S}\}\right)\end{array}$$

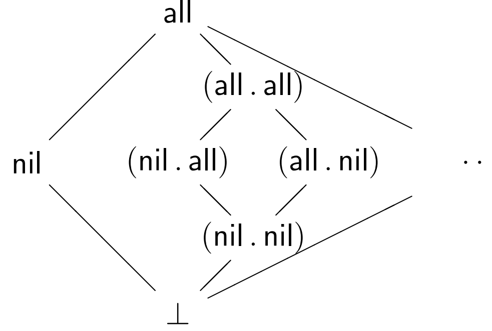
With these definitions $\tilde{\mathbf{V}}_\perp$ is a complete lattice with finite height (see [Rosendahl 1989]), and the functions α and γ are monotonic

$$\begin{array}{l}\forall \mathbf{x}_1, \mathbf{x}_2 \in \tilde{\mathbf{V}}\ .\ \mathbf{x}_1 \sqsupseteq \mathbf{x}_2 \Rightarrow \gamma(\mathbf{x}_1) \supseteq \gamma(\mathbf{x}_2) \\ \forall \mathbf{S}_1, \mathbf{S}_2 \subseteq \mathbf{V}\ .\mathbf{S}_1 \supseteq \mathbf{S}_2 \Rightarrow \alpha(\mathbf{S}_1) \sqsupseteq \alpha(\mathbf{S}_2)\end{array}$$

and continuous. The dyadic least upper bound is

$$\begin{array}{l}\mathsf{lub}(\mathbf{x}, \mathbf{y}) = \textbf{if } \mathbf{x} = \mathbf{y} \textbf{ then } \mathbf{x} \\ \qquad\qquad \textbf{else if } \mathbf{x} = \mathsf{all} \textbf{ or } \mathbf{y} = \mathsf{all} \textbf{ or} \\ \qquad\qquad\qquad \mathsf{atom}(\mathbf{x}) \textbf{ or } \mathsf{atom}(\mathbf{y}) \textbf{ then } \mathsf{all} \\ \qquad\qquad \textbf{else } \tilde{\mathsf{cons}}(\mathsf{lub}(\tilde{\mathsf{car}}(\mathbf{x}), \tilde{\mathsf{car}}(\mathbf{y})), \mathsf{lub}(\tilde{\mathsf{cdr}}(\mathbf{x}), \tilde{\mathsf{cdr}}(\mathbf{y})))\end{array}$$

where $\tilde{\mathsf{cons}}$, $\tilde{\mathsf{car}}$, and $\tilde{\mathsf{cdr}}$ are defined below.



Some elements from the complete lattice $\tilde{\mathbf{V}}_\perp$

Tilde-extensions Functions on S-expressions can be extended to functions on $\tilde{\mathbf{V}}$. We need such tilde-extensions of the basic operations in the complexity analysis. Let $\mathbf{f} : \mathbf{V}^* \to \mathbf{V}_\perp$ be a function on S-expressions. The induced function will be $\tilde{\mathbf{f}} : \tilde{\mathbf{V}}^* \to \tilde{\mathbf{V}}_\perp$.

$$\tilde{\mathbf{f}}(\mathbf{x}_1, \ldots, \mathbf{x_i}) = \alpha\left(\{\mathbf{f}(\mathbf{y}_1, \ldots, \mathbf{y_i}) \mid \mathbf{y_j} \in \gamma(\mathbf{x_j}), \mathbf{j} = 1, \ldots, \mathbf{i}\}\right)$$

The extensions for the basic operations are fairly obvious,

$$\begin{aligned}
\tilde{\mathsf{car}}(\mathbf{x}) &= \mathsf{all} \quad \textbf{if } \mathbf{x} = \mathsf{all} \text{ , } \textbf{else} \\
& \quad\; \mathbf{a} \quad\;\; \textbf{if } \mathbf{x} = \langle \mathbf{a}, \mathbf{b} \rangle \\
\tilde{\mathsf{cons}}(\mathbf{x}, \mathbf{y}) &= \langle \mathbf{x}, \mathbf{y} \rangle \\
\tilde{\mathsf{eq}}(\mathbf{x}, \mathbf{y}) &= \mathsf{all} \quad \textbf{if } \mathbf{x} = \mathsf{all} \vee \mathbf{y} = \mathsf{all} \text{ , } \textbf{else} \\
& \quad\; \mathsf{true} \quad \textbf{if } \mathbf{x} = \mathbf{y} \wedge \mathsf{atom}(\mathbf{x}) \text{ , } \textbf{else} \\
& \quad\; \mathsf{false}
\end{aligned}$$

Other basic operations can be extended in a similar way.

Functions from $\mathbf{V}$ to $\mathbb{N}_\perp$ can also be extended to functions from $\tilde{\mathbf{V}}$ to $\mathbb{N}_\perp^\infty$. Let

$$\mathbf{f} : \mathbf{V}^* \to \mathbb{N}_\perp$$

We then define

$$\tilde{\mathbf{f}} : \tilde{\mathbf{V}}^* \to \mathbb{N}_\perp^\infty$$

as

$$\tilde{\mathbf{f}}(\mathbf{x}_1, \ldots, \mathbf{x_i}) = \mathsf{max}\{\mathbf{f}(\mathbf{y}_1, \ldots, \mathbf{y_i}) \mid \mathbf{y_j} \in \gamma(\mathbf{x_j})\}$$

This extension will also be continuous.

Tilde-interpretation The complexity interpretation $\tilde{\mathsf{U}}$ of programs. The basic operations $\widetilde{\mathbf{basic}}$ are the extensions defined above and max takes the maximum of two numbers.

$$\tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{e}]\!] : (\tilde{\mathbf{V}}^{\mathbf{n}} \to (\tilde{\mathbf{V}}_{\perp} \times \mathbb{N}^{\infty}_{\perp}))^{\mathbf{k}} \to \tilde{\mathbf{V}}^{\mathbf{n}} \to (\tilde{\mathbf{V}}_{\perp} \times \mathbb{N}^{\infty}_{\perp})$$

$$\begin{aligned}
&\tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{c_i}]\!]\phi\nu = \langle \mathbf{const_i}, 1\rangle\\
&\tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{x_i}]\!]\phi\nu = \langle \nu{\downarrow}\mathbf{i}, 1\rangle\\
&\tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{a_i}(\mathbf{E}_1,\ldots,\mathbf{E_n})]\!]\phi\nu = \mathbf{let}\ \langle\alpha_{\mathbf{j}},\beta_{\mathbf{j}}\rangle = \tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{E_j}]\!]\phi\nu\\
&\qquad\qquad \mathbf{in}\ \langle \widetilde{\mathbf{basic}}_{\mathbf{i}}(\alpha_1,\ldots,\alpha_{\mathbf{n}}), 1+\beta_1+\cdots+\beta_{\mathbf{n}}\rangle\\
&\tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{f_i}(\mathbf{E}_1,\ldots,\mathbf{E_n})]\!]\phi\nu = \mathbf{let}\ \langle\alpha_{\mathbf{j}},\beta_{\mathbf{j}}\rangle = \tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{E_j}]\!]\phi\nu, \mathbf{j}=1,\ldots,\mathbf{n}\\
&\qquad\qquad \mathbf{in\ if\ some}\ \alpha_{\mathbf{j}} = \perp\ \mathbf{then}\ \langle\perp, \beta_1+\cdots+\beta_{\mathbf{n}}\rangle\\
&\qquad\qquad \mathbf{else\ let}\ \langle\alpha_0,\beta_0\rangle = \phi_{\mathbf{i}}(\alpha_1,\ldots,\alpha_{\mathbf{n}})\\
&\qquad\qquad \mathbf{in}\ \langle\alpha_0, 1+\beta_0+\beta_1+\cdots+\beta_{\mathbf{n}}\rangle\\
&\tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{if}\ \mathbf{E}_1\ \mathbf{then}\ \mathbf{E}_2\ \mathbf{else}\ \mathbf{E}_3]\!]\phi\nu =\\
&\quad \mathbf{case}\ \tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{E}_1]\!]\phi\nu\ \mathbf{of}\\
&\qquad \langle \mathsf{all}, \beta_1\rangle \Rightarrow \mathbf{let}\ \langle\alpha_2,\beta_2\rangle = \tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{E}_2]\!]\phi\nu\\
&\qquad\qquad \langle\alpha_3,\beta_3\rangle = \tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{E}_3]\!]\phi\nu\\
&\qquad\qquad \mathbf{in}\ \langle \mathsf{lub}(\alpha_2,\alpha_3), 1+\beta_1+\mathsf{max}(\beta_2,\beta_3)\rangle\\
&\qquad \langle \mathsf{true}, \beta_1\rangle \Rightarrow \mathbf{let}\ \langle\alpha_2,\beta_2\rangle = \tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{E}_2]\!]\phi\nu\\
&\qquad\qquad \mathbf{in}\ \langle\alpha_2, 1+\beta_1+\beta_2\rangle\\
&\qquad \langle \mathsf{false}, \beta_1\rangle \Rightarrow \mathbf{let}\ \langle\alpha_3,\beta_3\rangle = \tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{E}_3]\!]\phi\nu\\
&\qquad\qquad \mathbf{in}\ \langle\alpha_3, 1+\beta_1+\beta_3\rangle\\
&\qquad \langle _, \beta_1\rangle \Rightarrow \langle \mathsf{nil}, 1+\beta_1\rangle\\
&\quad \mathbf{end}
\end{aligned}$$

$$\tilde{\mathsf{U}}_{\mathbf{t}}[\![\mathbf{p}]\!] : (\tilde{\mathbf{V}}^{\mathbf{n}} \to (\tilde{\mathbf{V}}_{\perp} \times \mathbb{N}^{\infty}_{\perp}))^{\mathbf{k}}$$

$$\begin{aligned}
\tilde{\mathsf{U}}_{\mathbf{t}}[\![\ &\mathbf{f}_1(\mathbf{x}_1,\ldots,\mathbf{x_n}) = \mathbf{E}_1\\
&\vdots\\
&\mathbf{f_k}(\mathbf{x}_1,\ldots,\mathbf{x_n}) = \mathbf{E_k}]\!] = \mathsf{fix}(\lambda\,\phi.\ \langle \tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{E}_1]\!]\phi,\ldots,\tilde{\mathsf{E}}_{\mathbf{t}}[\![\mathbf{E_k}]\!]\phi\rangle)
\end{aligned}$$

Chapter 4

A Metalanguage for Abstract Interpretation

A fixpoint semantics is often used to describe the usual meaning of programs in a language. By doing this we represent computable entities (programs) as mathematical objects in cpo's.

The implementation of abstract interpretation is an example of an operation in the opposite direction. We define a fixpoint semantics and want to construct an interpreter which evaluates the meaning as described by the fixpoint semantics. This situation is also known from automatic compiler generation where one describes a semantics which is used to generate a compiler for the language described by the semantics.

When a fixpoint semantics is used to describe the meanings of programming languages there are no intrinsic restrictions on the language. True, the language must be well-defined both syntactically and semantically but the technique has been used on a wide variety of languages.

The class of cpo's and continuous functions, on the other hand, is much richer than the computable functions, and we may easily describe a fixpoint semantics which cannot be implemented. By restricting the metalanguage in which abstract interpretations are specified we may, however, force the functions defined in this way to be computable. Unfortunately such restrictions may be so restrictive as to make the metalanguage useless for program analysis.

Our aim in this chapter is to discuss systems of cpo's for which effective implementations are possible. The main problem in the implementation is to find fixpoints in domains. For a certain class of functions this can be done by a Y-combinator or a **letrec**-expression as known from many programming languages. As we shall see, this class is too small for many applications in abstract interpretation. To extend the class we introduce a method called *lazy fixpoint iteration* as a technique to find fixpoint in domains of finite height. The more general problem of defining classes of effective cpo's and a metalanguage which define exactly the computable elements is far beyond the scope of this thesis.

4.1 Computability

In abstract interpretation *termination* plays a central role. If we design a program analysis to be used in a compiler it will normally be required that it is guaranteed to terminate. This can be achieved by the condition that all cpo's in fixpoint iteration have finite height. If we write an interpreter for a programming language we do not expect termination to be guaranteed. Denotational semantics originally assumed a weaker termination criterion in that it used the untyped λ-calculus as a metalanguage [Stoy 1977]. Now denotational semantics often means domain-theoretic semantics and the metalanguage is the typed λ-calculus with fixpoint operations. The implementation (or computability) aspects of these kinds of semantics is the subject of this section.

4.1.1 Lambda-definability

The class of partial recursive functions is a precisely defined metalanguage of functions over the natural numbers. By Church's thesis this corresponds to the class of functions we can realise with computable algorithms. Kleene (see [Barendregt 1984]) showed that this class can be identified with the class of λ-definable functions: the functions which can be expressed as λ-terms using a proper encoding of the natural numbers. Scott constructed a domain-theoretic model for the λ-calculus and thereby gave a mathematical basis for defining semantics in the λ-calculus.

On the basis of this there could be reasons to consider the λ-calculus as a basis for abstract interpretation. Computability does not, however, imply termination so extra restrictions must be imposed if we want to use an abstract semantics in a compiler. Furthermore, the encoding of composite domains and solutions to recursive domain equations in the λ-calculus may be unnecessarily complicated.

4.1.2 Decidability

Fixpoints of continuous functions on domains with finite height can be found with a finite number of iterations. This is in essence a decidable logical system. The fixpoint iteration requires an (intentional) equality on the elements in the domain and that the continuous function is computable and total.

In this case the restriction that secures termination is placed on the domains rather than the functions. There may, however, be examples where it is difficult to ascertain that the domain has finite height. In a live variable analysis we may use the powerset of variable names as the abstract domain. This domain has infinite height but any one program will only contain a finite number of variables. Here, it is the use of the domain and not the structure of the domain that secures termination. [Cousot & Cousot 1977] introduced the concept of *widening* as a method to force fixpoint iteration in a domain of infinite height only to use a finite number of elements in a

sequence. The fixpoint iteration will terminate but it will not necessarily find the least fixpoint.

4.1.3 Effective domains

Even the simple domain $\mathbb{N} \to \mathbb{N}_\perp$ will contain more than just computable functions. The domain is not countable and there are only countably many partial recursive functions. This leads to the idea of a constructive domain theory; we may be able to define smaller domains which only contain the sequentially implementable functions. If such domains were used as a basis for a metalanguage for abstract interpretation all objects would, at least, be computable. Such *effective domains* (see [Phoa 1990]) are still an open research area and there are to the best of our knowledge no satisfactory solutions to the characterisation of such functions in the higher-order case.

4.1.4 Fixpoints

Fixpoints and recursive definitions play a central role in describing computability of functions. We have mentioned a number of different methods to give recursive definitions of functions:

Partial recursive functions A recursive function may be defined using the *least number operator* μ where

$$\mu\, \mathbf{n}[\mathbf{g}(\mathbf{n}_1, \ldots, \mathbf{n_k}, \mathbf{n})]$$

is the least number $\mathbf{n}$ such that $\mathbf{g}(\mathbf{n}_1, \ldots, \mathbf{n_k}, \mathbf{n}) = 0$.

Lambda calculus In the untyped λ-calculus [Barendregt 1984] it is possible to define a λ-expression $\mathbf{Y}$ such that for all λ-expressions $\mathbf{F}$ that $\mathbf{YF}$ and $\mathbf{F}(\mathbf{YF})$ are equivalent under β-convertibility.

$$\mathbf{Y} = \lambda\, \mathbf{f}.(\lambda\, \mathbf{x}.\mathbf{f}(\mathbf{x}(\mathbf{x})))(\lambda\, \mathbf{x}.\mathbf{f}(\mathbf{x}(\mathbf{x})))$$

and

$$\mathbf{Y} = (\lambda\, \mathbf{x}\, \lambda\, \mathbf{y}.\mathbf{y}(\mathbf{x}(\mathbf{x}(\mathbf{y}))))(\lambda\, \mathbf{x}\, \lambda\, \mathbf{y}.\mathbf{y}(\mathbf{x}(\mathbf{x}(\mathbf{y}))))$$

are examples of such fixpoint combinators.

Domain theory In domain theory the fixpoint operator can be written as

$$\mathsf{fix} = \lambda\,\mathbf{F}.\lim_{\mathbf{n}\to\infty}\mathbf{F}^{\mathbf{n}}(\bot)$$

where $\bot$ is the bottom element of the domain. The operator will find the fixpoint by re-evaluating the argument until convergence. Continuity of $\mathbf{F}$ guarantees that

$$\bot \sqsubseteq \mathbf{F}(\bot) \sqsubseteq \mathbf{F}(\mathbf{F}(\bot)) \sqsubseteq \cdots$$

is a chain (the ascending Kleene chain) and if the domain in question has finite height then the limit will be found after a finite number of iterations.

This is not in general a useful computational model as the objects in the chain may be infinite. As an example take the factorial function

$$\mathsf{fix}(\lambda\,\mathbf{f}.\,\lambda\,\mathbf{x}.\ \mathbf{if}\ \mathbf{x} = 1\ \mathbf{then}\ 1\ \mathbf{else}\ \mathbf{x}\cdot\mathbf{f}(\mathbf{x}-1))$$

If functions are described as lists of argument-result pairs the fixpoint iteration will look like this

$$\begin{array}{ll}
\bot & = [] \\
\mathbf{F}(\bot) & = [1 \to 1] \\
\mathbf{F}^2(\bot) & = [1 \to 1, 2 \to 2] \\
\mathbf{F}^3(\bot) & = [1 \to 1, 2 \to 2, 3 \to 6] \\
\vdots &
\end{array}$$

The fixpoint will be an infinite list but fortunately we only need a finite approximation to calculate the factorial of a number. More complex situations arise when higher-order functions are defined as fixpoints. Not only is the fixpoint an infinite list but the iteration steps may contain infinite elements as well.

Finite height domains For a finite domain we may use the ascending Kleene sequence to find the fixpoint. Informally the fixpoint iterator can be written as:

```
λF. begin var a, b;
      a := ⊥;
      b := F(a);
       while b ≠ a do
         a := b;
         b := F(a)
       end ;
       return (b)
    end
```

where the function $\mathbf{F}$ is reevaluated starting with $\bot$ until a fixpoint has been found.

Functional programming The representations of the domain theoretic fixpoint in a functional programming language will typically use a **letrec** expression to construct a circular environment. The factorial function can be defined as

$$\textbf{letrec } \mathbf{f} = \lambda\,\mathbf{x}.\ \textbf{if } \mathbf{x} = 1 \textbf{ then } 1 \textbf{ else } \mathbf{x} \cdot \mathbf{f}(\mathbf{x} - 1) \textbf{ in } \ldots$$

The fixpoint operation will in this case not perform the fixpoint iteration but it constructs an expression which, when applied, evaluates the fixpoint. Externally it does not matter whether a function is defined as a fixpoint or as a simple lambda expression: they can be carried around and applied in the same way.

4.1.5 The bottom element

The computability of the bottom element is a main difference between the implementation of fixpoint iteration for finite domains and recursive definitions in functional programming. If we describe the semantics of a programming language in domain theory we will normally use the bottom element to denote non-termination. The unsolvability of the halting problem further tells us that there are no general methods to transform the denotation of a program into an algorithm where the bottom element is a testable value. In the implementation of fixpoint iteration for domains with finite height the bottom value must be a simple testable value. We use *testable* in this context to mean that it has a finite representation and that an equivalence operation is defined (returns true or false) for this value.

These two views of the bottom element in a domain cannot easily be combined.

Example The following fixpoint equation over sets of variable names could be part of a dead-variable analysis.

$$\begin{aligned}\mathbf{S} = \{\text{“x”}\} &\cup (\textbf{if } \text{“x”} \in \mathbf{S} \textbf{ then } \{\text{“y”}, \text{“z”}\} \textbf{ else } \{\text{“y”}\}) \\ &\cup (\textbf{if } \text{“x”} \in \mathbf{S} \wedge \text{“z”} \in \mathbf{S} \textbf{ then } \{\text{“v”}\} \textbf{ else } \{\})\end{aligned}$$

The fixpoint iteration will start with the bottom element—the empty set and iterate until a fixpoint has been reached.

$$\begin{aligned}\mathbf{S}_0 &= \{\} \\ \mathbf{S}_1 &= \{\text{“x”}\} \\ \mathbf{S}_2 &= \{\text{“x”}, \text{“y”}, \text{“z”}\} \\ \mathbf{S}_3 &= \{\text{“x”}, \text{“y”}, \text{“z”}, \text{“v”}\} \\ \mathbf{S}_4 &= \{\text{“x”}, \text{“y”}, \text{“z”}, \text{“v”}\}\end{aligned}$$

The fixpoint has been reached when re-evaluation does not include new values.

It is not obvious how to express this fixpoint iteration as a recursive function definition. The set **S** can be expressed by its membership (or characteristic) function:

$$\mathbf{s} = \lambda\,\mathbf{a}.\ \textbf{if } \mathbf{a} \in \mathbf{S} \textbf{ then } \mathsf{true} \textbf{ else } \mathsf{false}$$

The question is whether it is possible to define a function **F** directly from the equation above such that the membership function **s** can be defined as the least fixpoint of **F**.

A simple answer is that the function

$$\mathbf{F}(\mathrm{s}) = \lambda\,\mathbf{a}.\ \mathbf{if}\ \mathbf{a} \in \{\text{“x”}, \text{“y”}, \text{“z”}, \text{“v”}\}\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ \mathsf{false}$$

but this cannot really be said to be defined directly from the equation.

As another attempt we may write

$$\begin{aligned}\mathbf{F}(\mathrm{s}) = \lambda\,\mathbf{a}.\ \mathrm{s}(\mathbf{a})\ \mathbf{or}\ (\mathbf{if}\ \mathrm{s}(\text{“x”})\ \mathbf{then}\ \mathbf{a} = \text{“y”}\ \mathbf{or}\ \mathbf{a} = \text{“z”}\ \mathbf{else}\ \mathbf{a} = \text{“y”})\\ \mathbf{or}\ (\mathbf{if}\ \mathrm{s}(\text{“x”})\ \mathbf{or}\ \mathrm{s}(\text{“y”})\ \mathbf{then}\ \mathbf{a} = \text{“z”}\ \mathbf{else}\ \mathsf{false})\end{aligned}$$

The least fixpoint of this function is the undefined function. We may, however, use a boolean domain with $\mathsf{false} \sqsubseteq \mathsf{true}$ with false as the bottom element, and let the **or** operation be a parallel **or**. This will make the fixpoint defined for members of **S** (with the value true) and undefined otherwise. It does not seem possible to improve on this result by lifting the domain.

4.1.6 Metalanguage

The general aim in this chapter is to find a computable subset of domain theory. We want to define a metalanguage which can be used to describe both elements in cpo's where fixpoints are implemented with **letrec**-expressions, and finite domains using fixpoint iteration.

Abstract interpretations uses frequently a more varied typestructure than denotational semantics. Finite lattices can be useful to give an approximate or abstract description of the computational behaviour. The expense of using these other domains is that the semantics cannot be interpreted directly as a lambda expression.

4.2 A simple typesystem

To start with we will examine the implementation of elements in a simple system of cpo's.

Consider cpo's built from three base domains $\mathbb{N}_\perp$, $\mathbb{B}_\perp$, and $\mathbb{A}_\perp$ as introduced in section 2.1, using function domains ($\rightarrow$), cartesian products ($\times$), separated sum ($+$), and recursive function definitions. In these domains we will only attempt to implement elements which have a finite description in a metalanguage containing the following operations.

Base domains

Constants, equality operation (eq), addition on numbers ($\mathsf{add} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}_\perp$), plus possibly some other algebraic and boolean operations.

Function domain
λ-abstraction, application, and use of parameters .

Cartesian product
Pairing and selection (pair, fst, and snd).

Separated sum
isl, isr, inl, inr, outl, outr.

Recursive domain definition
Injection map.

For all domains D
Conditional: $\mathsf{if} : \mathbb{B}_\bot \times \mathbf{D} \times \mathbf{D} \to \mathbf{D}$.
Fixpoint: $\mathsf{fix} : (\mathbf{D} \to \mathbf{D}) \to \mathbf{D}$.

The results on implementation are rather obvious: essentially we can implement anything that only uses finite, non-bottom values. This class of cpo's with elements expressed in the metalanguage is identical to what is normally used in denotational semantics (see [Gordon 1979], [Schmidt 1986], [Stoy 1977]). This metalanguage is also equivalent to the TML language [Nielson 1987] used in the PSI project where it has been implemented in ML. Implementation of Scott-Strachey style denotational semantics has been made in [Mosses 1979] in the framework of a compiler-generator and in [Schmidt 1982] in the form of an interpreter.

4.2.1 Implementation language

It seems easier to use a lazy higher-order language as an implementation language, especially for domains where the bottom element will be represented by nontermination. It is possible to use a strict language like ML as in [Nielson 1987] but in most situations this will require that values are protected by lambda abstractions. We will not use a typed implementation language as the values that are implemented are strongly typed and a type system cannot add more structure.

The language we will use is a lazy version of Lisp in a notation comparable to LML and similar languages. There are three types of values:

- Base values are integers, boolean values, and strings.
- Pairs are tuples of values.
- Functions are mappings of values to values.

Expressions are

- Constants: integers, boolean values or strings.

- Pairing and selection. For expressions $\mathbf{e}_1$ and $\mathbf{e}_2$, $\langle \mathbf{e}_1, \mathbf{e}_2 \rangle$ will evaluate to a pair with the value of $\mathbf{e}_1$ as the first field and the value of $\mathbf{e}_2$ as the second. $\mathbf{e}{\downarrow}1$ will select the first field of the pair in the value of $\mathbf{e}$, and $\mathbf{e}{\downarrow}2$ will select the second field.

- Application and lambda abstraction.

- Conditional. The expression $\mathbf{if}\ \mathbf{e}_1\ \mathbf{then}\ \mathbf{e}_2\ \mathbf{else}\ \mathbf{e}_3$ will return the value of $\mathbf{e}_2$ or $\mathbf{e}_3$ depending on the value of $\mathbf{e}_1$.

- Equality predicate. $\mathbf{e}_1 = \mathbf{e}_2$ returns **true** if both $\mathbf{e}_1$ and $\mathbf{e}_2$ return the same base value, or if both return tuples and $\mathbf{e}_1{\downarrow}1 = \mathbf{e}_2{\downarrow}1$ and $\mathbf{e}_1{\downarrow}2 = \mathbf{e}_2{\downarrow}2$.

- Recursive definition. The expression $\mathbf{letrec}\ \mathbf{n} = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2$ will return the value of $\mathbf{e}_2$ where any occurrence of $\mathbf{n}$ in $\mathbf{e}_1$ or $\mathbf{e}_2$ has the value of $\mathbf{e}_1$.

4.2.2 Fixpoints

The fixpoint operations can for any domain $\mathbf{D}$ be implemented as

$$\mathsf{fix}_{\mathbf{D}} = \lambda \mathbf{f}.\ \mathbf{letrec}\ \mathbf{x} = \mathbf{f}(\mathbf{x})\ \mathbf{in}\ \mathbf{x}$$

The proof is by induction over the structure of the type τ. Let $\mathbf{D}$ be a (flat) base domain and $\mathbf{f} : \mathbf{D} \to \mathbf{D}$ a function. The claim is that $\mathsf{fix}(\mathbf{f}) = \mathbf{f}(\bot)$ as any use of the argument to $\mathbf{f}$ will result in the fixpoint being $\bot$. Let $\mathbf{a}$ be given as $\mathbf{f}(\bot)$. If $\mathbf{a} = \bot$ then $\mathbf{a}$ is the fixpoint, else let $\mathbf{f}(\mathbf{a}) = \mathbf{b}$, then monotonicity gives $\mathbf{a} \sqsubseteq \mathbf{b}$. By definition this means that $\mathbf{a} = \bot \vee \mathbf{a} = \mathbf{b}$ which means that $\mathbf{a}$ is the fixpoint.

4.2.3 Lifted domains

We have not included lifted domains in the class of cpo's we want to implement. The reason for this is we can not make an interesting implementation as long as bottom is not testable. For a domain $\mathbf{D}$ we should for the domain $\mathbf{D}_\bot$ implement

$$\begin{aligned} \mathsf{def} &: \mathbf{D}_\bot \to \mathbb{B}_\bot \\ \mathsf{up} &: \mathbf{D} \to \mathbf{D}_\bot \\ \mathsf{down} &: \mathbf{D}_\bot \to \mathbf{D} \end{aligned}$$

The function def will return true for values in $\mathbf{D}$ and undefined ($\bot$) otherwise. This means that we can construct a lifted domain but we cannot test whether a value is "lifted" (member of $\mathbf{D}$ and not $\bot$).

4.3 Finite domains

Let us consider a finite domain $\mathbf{D}$

$$\mathbf{D} = \{\mathbf{a}_1, \ldots, \mathbf{a_n}\}$$

The domain must be (partially) ordered with a bottom element $\mathbf{a_k}$ for some $\mathbf{k} \in \{1, \ldots, \mathbf{n}\}$. The ordering is not necessarily flat. In this case we do not want to model the bottom value as undefined, so in order not to confuse these two concepts we will use $\mathbf{a_k}$ to denote the bottom element in $\mathbf{D}$ rather than $\perp_\mathbf{D}$.

To find the fixpoint of continuous functions over this domain the usual method is to apply the function to the bottom element repeatedly until the same value has been achieved twice. Notice, by the way, that the ordering is without importance for the fixpoint iteration as long as the bottom element is given and the function is continuous.

To express the fixpoint iteration we need to extend the implementation language with an assignment operation.

Assignment To define the fixpoint of a function over this domain we will extend the implementation language with an imperative feature. The

```
begin
   ...
   return (⟨exp⟩)
end
```

expression returns the value of the expression after **return** and the commands (...) are performed if any part of the result is needed. The commands in the **begin** expression are written in a syntax similar to Pascal. They may include assignments, **if**-statements, and **while**-statements. For the semantics we will assume that assignments to local variables only are local and assignments to global variables are global. The assignment operation in a lazy language is a nasty concept and can not be recommended for general programming. In this case it is only used in a rather restricted manner to describe an algorithm. When the fixpoint operation is used the imperative element is hidden and only the fixpoint is available.

```
fix_D = λx.  begin var a, b;
               a := a_k;
               b := x(a);
                while b ≠ a begin
                  a := b;
                  b := x(a)
                end ;
                return (b)
             end
```

We have assumed here that there is an equality operation defined on the domain.

Power set For a finite set $\mathbf{D}$ the power set $\mathbb{P}(\mathbf{D})$ is also finite. Assuming $\mathbf{D}$ can be represented in the implementation language, elements in $\mathbb{P}(\mathbf{D})$ can be represented as lists of values from $\mathbf{D}$. The fixpoint iteration from above can also be used for sets except that set equality should be used instead and the bottom value is the empty set. As with our approach, pending analysis and minimal function graphs have so far only been defined in the first order case.

Frontier algorithm Fixpoint iteration in abstract interpretation can be costly and several methods have been proposed to make implementation practical. For strictness analysis and other interpretations over domains with only a few elements the *frontier* algorithm seems to be the best bet. The method was proposed by Peyton Jones and Clark ([Clack & Jones 1985], [Jones 1987b]) and several improvements and extensions have since been published ([Martin & Hankin 1987], [Jones & Clack 1987] [Hunt 1989]).

Pending analysis The method we will describe in the rest of this chapter bears some resemblance to *pending analysis* [Young & Hudak 1986] and minimal function graphs [Jones & Mycroft 1986]. Pending analysis is a method for finding fixpoints of recursive monotone boolean functions. It uses the observation that at the second recursive call to such a function it is safe to return bottom. The approach described here can be used for mutually recursive functions over domains of finite height, even when the height is not known beforehand.

4.3.1 Composite domains

There are no simple ways to extend a fixpoint operation on a finite domain to composite domains. The obvious way to extend the above fixpoint operation for $\mathsf{fix}_{\mathbf{D}}$ to a cartesian product $\mathsf{fix}_{\mathbf{D}\times\mathbf{D}}$ is to make two simultaneous iterations for fixpoints.

$$
\begin{aligned}
\mathsf{fix}_{\mathbf{D}\times\mathbf{D}} = \lambda\,\mathbf{x}.\ & \textbf{begin var } \mathbf{x}_1, \mathbf{x}_2, \mathbf{y}_1, \mathbf{y}_2; \\
& \quad \mathbf{x}_1 := \mathbf{a_k}; \\
& \quad \mathbf{x}_2 := \mathbf{a_k}; \\
& \quad \mathbf{y}_1 := \mathbf{x}(\mathbf{x}_1, \mathbf{x}_2)\downarrow 1; \\
& \quad \mathbf{y}_2 := \mathbf{x}(\mathbf{x}_1, \mathbf{x}_2)\downarrow 2; \\
& \quad \textbf{while } (\mathbf{x}_1 \neq \mathbf{y}_1) \wedge (\mathbf{x}_2 \neq \mathbf{y}_2) \textbf{ begin} \\
& \qquad \mathbf{x}_1 := \mathbf{y}_1; \\
& \qquad \mathbf{x}_2 := \mathbf{y}_2; \\
& \qquad \mathbf{y}_1 := \mathbf{x}(\mathbf{x}_1, \mathbf{x}_2)\downarrow 1; \\
& \qquad \mathbf{y}_2 := \mathbf{x}(\mathbf{x}_1, \mathbf{x}_2)\downarrow 2;
\end{aligned}
$$

$$\begin{array}{l}\quad\textbf{end} ;\\ \quad\textbf{return}\ \langle \mathbf{x}_1, \mathbf{x}_2 \rangle\\ \textbf{end}\end{array}$$

The problem with this scheme, from a computational point of view, is that it will find both fields in the result even if only one is needed.

It is possible to define a fixpoint operation that only finds a fixpoint if it is needed. It is known as Bekic theorem ([Nielson & Nielson 1988] contains a version for a cartesian product of $\mathbf{n}$ domains).

$$\begin{array}{l}\mathsf{fix}_{\mathbf{D}\times\mathbf{D}} = \lambda\,\mathbf{x}.\ \textbf{letrec}\\ \qquad \mathbf{a}_1 = \mathsf{fix}_{\mathbf{D}}\,\lambda\,\mathbf{x}_1.\mathbf{x}(\mathbf{x}_1, \mathbf{a}_2(\mathbf{x}_1))\downarrow 1\\ \qquad \mathbf{a}_2 = \lambda\,\mathbf{x}_1.\,\mathsf{fix}_{\mathbf{D}}\,\lambda\,\mathbf{x}_2.\mathbf{x}(\mathbf{x}_1, \mathbf{x}_2)\downarrow 2\\ \quad \textbf{in}\ \langle \mathbf{a}_1, \mathbf{a}_2(\mathbf{a}_1)\rangle\end{array}$$

Computationally this method is very unattractive. If the first field requires $\mathbf{n}$ iterations and uses the second field then the fixpoint for the second field will be found $\mathbf{n}+1$ times. This merely shows that it is possible to define a fixpoint operation with the desired property. We will in the next section present an algorithm which removes most of the re-evaluation.

4.4 Lazy fixpoint iteration

The Y-combinator gives a good model for a fixpoint operator. It constructs a recipe for the evaluation of the fixpoint and only if we need the result will we compute an approximation to the fixpoint. In this section we introduce the evaluation strategy for the lazy fixpoint iteration. We start with a special case which should illustrate the underlying idea. The example is fixpoint iteration for first-order functions on flat domains. Later in the section the principles will be extended to non-flat domains.

4.4.1 Memo-function

Consider a function $\mathbf{f} : \mathbf{S} \to \mathbf{S}_\perp$ over a set $\mathbf{S}$ where the function $\mathbf{f}$ is defined as the (least) fixpoint of a functional $\mathbf{F} : (\mathbf{S} \to \mathbf{S}_\perp) \to (\mathbf{S} \to \mathbf{S}_\perp)$.

The fixpoint can be found as

$$\mathbf{f} = \mathsf{fix}_{\mathbf{S}\to\mathbf{S}_\perp}(\mathbf{F})$$

where

$$\mathsf{fix}_{\mathbf{S}\to\mathbf{S}_\perp} = \textbf{letrec}\ \mathbf{f} = \mathbf{F}(\mathbf{f})\ \textbf{in}\ \mathbf{f}$$

Assume now that we want the function to be implemented using a *memo-function* such that repeated calls to $\mathbf{f}$ with the same arguments do not require re-evaluation.

We can obtain this by constructing a data structure containing a list of argument-result pairs.

$$
\begin{aligned}
\mathsf{fix}_{\mathbf{S}\to\mathbf{S}_\perp}(\mathbf{F}) = \ & \mathbf{begin\ var\ l};\\
& \mathbf{l} := [];\\
& \mathbf{return}\ (\ \mathbf{letrec}\\
& \qquad\qquad \mathbf{f}' = \mathbf{F}(\mathbf{f}'');\\
& \qquad\qquad \mathbf{f}'' = \lambda\,\mathbf{x}.\,\mathsf{take}(\mathbf{x},\mathbf{l},\mathbf{f}')\\
& \qquad\quad \mathbf{in}\ \mathbf{f}'')\\
& \mathbf{begin}
\end{aligned}
$$

where the function take checks whether the function has been evaluated already for the given argument. If so it returns the previous value, otherwise it evaluates the result and includes the argument-result pair in the data structure.

$$
\begin{aligned}
& \mathsf{take}(\mathbf{x},\mathbf{l},\mathbf{f}) =\\
& \quad \mathbf{case}\ \mathsf{assoc}(\mathbf{x},\mathbf{l})\ \mathbf{of}\\
& \qquad [\,]\ \Rightarrow \quad \mathbf{let}\ \mathbf{y} = \mathbf{f}(\mathbf{x})\\
& \qquad\qquad\quad\ \mathbf{in\ begin}\ \mathbf{l} := \langle \mathbf{x},\mathbf{y}\rangle :: \mathbf{l};\\
& \qquad\qquad\qquad\qquad \mathbf{return}\ (\mathbf{y})\\
& \qquad\qquad\quad\ \mathbf{end}\ ;\\
& \qquad \langle \mathbf{a},\mathbf{y}\rangle\ \Rightarrow\ \mathbf{y}\\
& \quad \mathbf{end}
\end{aligned}
$$

The function assoc is defined as in Lisp [Foderaro, Sklower & Layer 1983]. It will implicitly use the equality on $\mathbf{S}$. :: is an infix pairing operation and [] the empty list.

The use of memo-functions changes the evaluation strategy to eager evaluation of arguments. This does not necessarily imply that the function is strict, since we are dealing with finite maps and finite domains where the bottom element is testable.

This implementation is in fact a realisation of the minimal function graph interpretation ([Jones & Mycroft 1986]). If the assoc operator is substituted with a function which only tests with eq instead of equal we will implement lazy memo-functions as described in [Hughes 1985].

Correctness Throughout the fixpoint iteration, $\mathbf{l}$ has the property

$$\forall\,\mathbf{x},\mathbf{y}.\,\mathsf{assoc}(\mathbf{x},\mathbf{l}) = \langle \mathbf{x},\mathbf{y}\rangle \Rightarrow \mathbf{f}(\mathbf{x}) = \mathbf{y}$$

from this it follows that

$$\mathsf{take}(\mathbf{x},\mathbf{l},\mathbf{f}) = \mathbf{f}(\mathbf{x})$$

and

$$
\begin{aligned}
& \mathbf{f}'' = \lambda\,\mathbf{x}.\mathbf{f}'(\mathbf{x}) = \mathbf{f}'\\
& \mathbf{f}' = \mathbf{F}(\mathbf{f}'') = \mathbf{F}(\mathbf{f}')
\end{aligned}
$$

So the result will be the same as in the simple definition of the fixpoint. □

4.4.2 Function domain

It is tempting to extend the memo-function idea to fixpoints over other domains. The fixpoint operator should just create a data structure or an expression and only when parts of the fixpoint is needed will it be evaluated or extracted from the structure. For cartesian products one can ask for the first or the second field and for functions the request is indexed by the argument value. This method can be used to achieve mutually recursive functions with a minimal function graph interpretation.

Unfortunately the method does not generalise easily to finite lattices, as it relies on the range of the function being a flat domain. Nevertheless it is possible to construct a memo-function implementation for functions on non-flat domains. This will be described later but first we discuss evaluation strategies for fixpoints.

Breadth-first iteration Let $\tau = \mathbf{D} \rightarrow \mathbf{E}$ where $\mathbf{D}$ is finite and $\mathbf{E}$ has finite height and $\mathbf{b}$ is the bottom element of $\mathbf{E}$. Let $\mathbf{F} : \tau \rightarrow \tau$ be a continuous functional. The least fixpoint of $\mathbf{F}$ can be found as the limit of the ascending chain

$$
\begin{aligned}
&\mathbf{f}_0 = \lambda\, \mathbf{x}.\mathbf{b} \\
&\mathbf{f}_1 = \mathbf{F}(\mathbf{f}_0) \\
&\vdots \\
&\mathbf{f}_\mathbf{i} = \mathbf{F}(\mathbf{f}_{\mathbf{i}-1}) \\
&\vdots \\
&\mathbf{f} = \bigsqcup \langle \mathbf{f}_\mathbf{i} \rangle_{\mathbf{i} \in \mathbb{N}}
\end{aligned}
$$

This is in effect a breadth-first iteration. Each $\mathbf{f}_\mathbf{i}$ can be seen as a list of argument-result pairs and for $\mathbf{f}_\mathbf{i}$ the recursion is truncated below the $\mathbf{i}^{\text{th}}$ level. From a computational point of view the method is impractical because the fixpoint is found for all arguments at the same time even though some of the arguments may not be needed.

Even though we require $\mathbf{D}$ to be a finite domain it may be the subdomain of an infinite domain and the choice of subdomain depends on other factors. As an example $\mathbf{D}$ may be the set of variable names which is a subset of the set of strings. The actual variable names depend on the program to be analysed but we know they constitute a finite set. Recursively defined domains give a similar situation. We may only be interested in lists of length less or equal to $\mathbf{n}$ where $\mathbf{n}$ is the highest number of arguments to a given function in the program to be analysed. If the elements of the list belong to finite domains the domain of lists with maximum length $\mathbf{n}$ is also finite.

If we want to evaluate $\mathbf{f}(\mathbf{d})$ for $\mathbf{d} \in \mathbf{D}$ then it is not enough to find the first $\mathbf{i}$ such that $\mathbf{f}_\mathbf{i}(\mathbf{d}) = \mathbf{f}_{\mathbf{i}+1}(\mathbf{d})$. The problem (called a *plateau*) is discussed in detail by [Clack & Jones 1985] and [Jones 1987b]. It is necessary to find the fixpoint for $\mathbf{d}$ as well as the arguments used during the evaluation of $\mathbf{f}(\mathbf{d})$.

Depth-first iteration Instead of the breadth-first iteration we will define a depth-first evaluation strategy which remembers the arguments used under the evaluation. We define a family of functions $\mathbf{g_i} : \mathbb{P}(\mathbf{D}) \times \mathbf{D} \rightarrow \mathbf{E}$ which gives an approximation to function $\mathbf{f}$ where we at most will perform $\mathbf{i} - 1$ reevaluations of $\mathbf{F}$ for the arguments supplied as the first arguments and $\mathbf{i}$ reevaluations for the second argument.

$$
\begin{aligned}
&\mathbf{g}_0(\mathbf{S}, \mathbf{x}) = \mathbf{b} \\
&\mathbf{g}_1(\mathbf{S}, \mathbf{x}) = \mathbf{if}\ \mathbf{x} \in \mathbf{S}\ \mathbf{then}\ \mathbf{g}_0(\varnothing, \mathbf{x}) \\
&\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{F}(\lambda\, \mathbf{y}.\mathbf{g}_1(\mathbf{S} \cup \{\mathbf{x}\}, \mathbf{y}))(\mathbf{x}) \\
&\vdots \\
&\mathbf{g_i}(\mathbf{S}, \mathbf{x}) = \mathbf{if}\ \mathbf{x} \in \mathbf{S}\ \mathbf{then}\ \mathbf{g_{i-1}}(\varnothing, \mathbf{x}) \\
&\qquad\qquad\qquad\quad \mathbf{else}\ \mathbf{F}(\lambda\, \mathbf{y}.\mathbf{g_i}(\mathbf{S} \cup \{\mathbf{x}\}, \mathbf{y}))(\mathbf{x})
\end{aligned}
$$

and define

$$\mathbf{f'_i} = \lambda\, \mathbf{x}.\mathbf{g_i}(\varnothing, \mathbf{x})$$

Correctness We will now prove that

$$\bigsqcup \langle \mathbf{f_i} \rangle_{\mathbf{i} \in \mathbb{N}} = \bigsqcup \langle \mathbf{f'_i} \rangle_{\mathbf{i} \in \mathbb{N}}$$

using fixpoint induction. We will prove that for any $\mathbf{i}$ that

$$\mathbf{f_i} \sqsubseteq \mathbf{f'_i} \sqsubseteq \mathbf{f_{i*|D|}}$$

where $|\mathbf{D}|$ is the number of elements in $\mathbf{D}$.

For $\mathbf{i} = 0$ this is obvious. By induction

$$
\begin{aligned}
\mathbf{f_i}(\mathbf{x}) &= \mathbf{F}(\mathbf{f_{i-1}})(\mathbf{x}) \\
&\sqsubseteq \mathbf{F}(\mathbf{f'_{i-1}})(\mathbf{x}) \\
&= \mathbf{F}(\lambda\, \mathbf{y}.\mathbf{g_{i-1}}(\varnothing, \mathbf{y}))(\mathbf{x}) \\
&\sqsubseteq \mathbf{F}(\lambda\, \mathbf{y}.\mathbf{g_i}(\{\mathbf{x}\})(\mathbf{x}) \\
&= \mathbf{g_i}(\varnothing, \mathbf{x}) \\
&= \mathbf{f'_i}(\mathbf{x})
\end{aligned}
$$

For the second part we will prove that

$$\lambda\, \mathbf{x}.\mathbf{g_i}(\mathbf{S}, \mathbf{x}) \sqsubseteq \mathbf{f_{|D|*(i-1)+|D-S|}}$$

This is true for $\mathbf{i} - 1$ and $\mathbf{S} = \mathbf{D}$. We will prove it by induction over $|\mathbf{D} - \mathbf{S}|$ using the fact that $\mathbf{g_i}(\varnothing, \mathbf{x}) = \mathbf{g_{i+1}}(\mathbf{D}, \mathbf{x})$.

Assume $\mathbf{x}_1 \in \mathbf{S}$ and that the inequality is proved for $\mathbf{i}$ and $\mathbf{S}$.

For $\mathbf{g_i}(\mathbf{S} - \{\mathbf{x}_1\}, \mathbf{x})$ we find if $\mathbf{x} \in (\mathbf{D} - \mathbf{S}) \cup \{\mathbf{x}\}$,

$$\begin{aligned}\mathbf{g_i}(\mathbf{S} - \{\mathbf{x}_1\}, \mathbf{x}) &= \mathbf{F}(\lambda\, \mathbf{y}.\mathbf{g_i}(\mathbf{S}, \mathbf{y}))(\mathbf{x}) \\ &\sqsubseteq \mathbf{F}(\mathbf{f}_{|\mathbf{D}|*(\mathbf{i}-1)+|\mathbf{D}-\mathbf{S}|}) \\ &= \mathbf{f}_{|\mathbf{D}|*(\mathbf{i}-1)+|\mathbf{D}-\mathbf{S}|+1}\end{aligned}$$

If $\mathbf{x} \notin (\mathbf{D} - \mathbf{S}) \cup \{\mathbf{x}\}$,

$$\begin{aligned}\mathbf{g_i}(\mathbf{S} - \{\mathbf{x}_1\}, \mathbf{x}) &= \mathbf{g_{i-1}}(\varnothing, \mathbf{x}) \\ &\sqsubseteq \mathbf{f}_{|\mathbf{D}|*(\mathbf{i}-1)} \\ &\sqsubseteq \mathbf{f}_{|\mathbf{D}|*(\mathbf{i}-1)+|\mathbf{D}-\mathbf{S}|+1}\end{aligned}$$

□

The proof shows the advantage of the depth-first evaluation strategy. The breadth-first strategy may require $|\mathbf{D}|$ times as many reevaluations of $\mathbf{F}$.

Algorithm The algorithm presented below is essentially a realisation of the depth-first iteration strategy. There are two minor optimisations. Firstly, the strategy has been combined with a memo-function concept such that once a fixpoint has been found for any arguments then the function will not be re-evaluated for these values at later calls. Secondly, the termination criterion for the iteration has been changed from $\mathbf{g_{i-1}} = \mathbf{g_i}$ to: the values in $\mathbf{g_{i-1}}$ used when evaluating $\mathbf{g_i}$ must be the same as the values found by $\mathbf{g_i}$. This means that the iteration may stop with $\mathbf{g}_1$ even when $\mathbf{g}_1 \neq \mathbf{g}_0$ if there are no circular definitions of values. This is the case when the range is a flat domain and, except for some bookkeeping, the algorithm will then be equivalent to the memo-function algorithm.

```
fix_{D→E}(F) =  begin var l1, l2, l3;
                   l1 = [];
                   l2 = [];
                   l3 = [];
                   return ( letrec  f' = F(f'');
                                    f'' = λ x. take(x, l1, l2, l3, f')
                            in f'')
                end
```

The function take will now use three lists of argument-result pairs. The lists and their meanings are:

$\mathbf{l}_1$: arguments under evaluation with their current value
$\mathbf{l}_2$: found fixpoint values
$\mathbf{l}_3$: used values while under evaluation

The list $\mathbf{l}_1$ represents the function $\mathbf{g_i}$, and $\mathbf{l}_3$ the function $\mathbf{g_{i-1}}$, except that it only contains the values of $\mathbf{g_{i-1}}$ used under the evaluation of $\mathbf{g_i}$.

```
take(x, l1, l2, l3, f) =
    if assoc(x, l2) ≠ nil
    then snd(assoc(x, l2))
    else if l1 = [] then
    begin  var y, l4
           l1 := [⟨x, b⟩];
           y := f(x);
           l1 := ⟨x, y⟩ :: l1;
           iterate(l1, l2, l3, f);
           return (snd(assoc(x, l2)))
    end else
    case assoc(x, l1) of
        nil  ⇒ begin  var y;
                       l1 := ⟨x, b⟩ :: l1;
                       y := f(x);
                       l1 := ⟨x, y⟩ :: l1;
               end
        ⟨x, y⟩  ⇒ begin  l3 := ⟨x, y⟩ :: l3;
                          return (y)
                  end
    end
```

and

```
iterate(l1, l2, l3, f) =
    begin  var z, l4;
           while l1 ⊉ l3 begin
              l4 := l1;
              l3 := [];
              forall ⟨x, y⟩ in l4 begin
                 z := f(x);
                 l1 := ⟨x, z⟩ :: l1
              end ;
           end ;
           l3 = [];
           l2 = append(l1, l2);
           l1 = [];
```

end

and

$$\mathbf{l}_1 \supseteq \mathbf{l}_3$$
$$\Updownarrow$$
$$\mathsf{assoc}(\mathbf{x}, \mathbf{l}_1) = \langle \mathbf{x}, \mathbf{y} \rangle \text{ whenever } \mathsf{assoc}(\mathbf{x}, \mathbf{l}_3) = \langle \mathbf{x}, \mathbf{y} \rangle$$

Each time a new value is required we evaluate the function (in iterate) until the results of this and dependent values do not change. The results for these values are then included in the list $\mathbf{l}_2$. No further fixpoint iteration is required if any of these values are required at a later stage.

4.4.3 Fixpoint iteration for other domains

Let us return to the example with a cartesian product of two finite domains. The fixpoint operation $\mathsf{fix}_{\mathbf{D}\times\mathbf{D}}$ can now be implemented using $\mathsf{fix}_{2\to\mathbf{D}}$ where 2 is a twopoint set.

Let

$$\begin{aligned}
&\mathsf{decode} : (2 \to \mathbf{D}) \to (\mathbf{D} \times \mathbf{D}) \\
&\mathsf{decode}(\mathbf{x}) = \langle \mathbf{x}(1), \mathbf{x}(2) \rangle \\
&\mathsf{encode} : (\mathbf{D} \times \mathbf{D}) \to (2 \to \mathbf{D}) \\
&\mathsf{encode}(\mathbf{x}) = \lambda\, \mathbf{y}.\ \mathbf{if}\ \mathbf{y} = 1\ \mathbf{then}\ \mathsf{fst}(\mathbf{x})\ \mathbf{else}\ \mathsf{snd}(\mathbf{x})
\end{aligned}$$

then

$$\mathsf{fix}_{\mathbf{D}\times\mathbf{D}} = \lambda\, \mathbf{x}.\, \mathsf{decode}(\mathsf{fix}_{2\to\mathbf{D}}(\lambda\, \mathbf{y}.\, \mathsf{encode}(\mathbf{x}(\mathsf{decode}(\mathbf{y})))))$$

The same method can be used for the cartesian product of two different domains except that the bottom element ($\mathbf{b}$) used in the function take now depends on the argument. If we want to define $\mathsf{fix}_{\mathbf{D}_1\times\mathbf{D}_2}$ using $\mathsf{fix}_{2\to(\mathbf{D}_1+\mathbf{D}_2)}$ where $\mathbf{D}_1$ has bottom $\mathbf{b}_1$ and $\mathbf{D}_2$ has bottom $\mathbf{b}_2$ then all occurrences of $\mathbf{b}$ in the function take should be changed to

$$\mathbf{if}\ \mathbf{x} = 1\ \mathbf{then}\ \mathsf{inl}(\mathbf{b}_1)\ \mathbf{else}\ \mathsf{inr}(\mathbf{b}_2)$$

In a similar fashion the fixpoint operator $\mathsf{fix}_{\mathbf{A}\to(\mathbf{B}\to\mathbf{C})}$ where $\mathbf{A}$ and $\mathbf{B}$ are finite and $\mathbf{C}$ has finite height, can be found using $\mathsf{fix}_{(\mathbf{A}\times\mathbf{B})\to\mathbf{C}}$.

It is more difficult to define $\mathsf{fix}_{(\mathbf{A}\to\mathbf{B})\to\mathbf{C}}$ even though the functions defined by the lazy fixpoint iteration algorithm will be represented by list structures. We will leave this as an open question. It should be possible to define a fixpoint iteration which only evaluates those arguments of type $\mathbf{A} \to \mathbf{B}$ needed by the later computation. It will however require more bookkeeping.

This restriction seems related to the requirement of "contravariantly pure" in [Nielson 1988] or "level-preserving" in [Nielson 1989]. In this case, however, we only impose the restrictions if we want to implement the interpretations. The restrictions are on the implementations and not the specifications.

4.5 A domain-theoretic language

When constructing implementations for elements in cpo's the main problem is to find fixpoints of functions.

4.5.1 The simple type structure

For a certain class of cpo's we can find fixpoints by **letrec** expressions in the implementation language. These cpo's can be described by a type system:

$$\begin{array}{rl} \tau_{\mathbf{s}} ::= & \mathbf{integer} \mid \mathbf{bool} \mid \mathbf{string} \\ \mid & \tau_{\mathbf{s}} \rightarrow \tau_{\mathbf{s}} \\ \mid & \tau_{\mathbf{s}} \times \tau_{\mathbf{s}} \\ \mid & \tau_{\mathbf{s}} + \tau_{\mathbf{s}} \\ \mid & \mathbf{rec}\ \mathbf{X}.\tau_{\mathbf{s}} \\ \mid & \mathbf{X} \end{array}$$

We can construct the fixpoint operator for domains with these types but results will only be defined if computations only require finite non-bottom values.

4.5.2 Finite height domains

We have described a method to find fixpoints on a domain $\mathbf{D}$ of finite height and the method was extended to functions in $\mathbf{S} \rightarrow \mathbf{D}$ where $\mathbf{S}$ is a finite set. The fixpoint operation must have an equality operation operation eq for the domain (also defined for bottom). It is necessary for the fixpoint operator to know which element in the domain is the bottom element. The rest of the ordering is only important in proof of well-definedness of fixpoints (proof of continuity and finite height). We can therefore specify a finite height domain as

$$\mathbf{domain}\ \{\mathbf{a}_1, \ldots, \mathbf{a_n}\}\ \mathbf{with\ bottom}\ \mathbf{a_k}$$

The powerset construction is a useful method to construct domains of finite height. For a finite set $\mathbf{S}$ the powerset can be specified as

$$\mathbf{power\ S}$$

where the bottom element is the empty set. We may define a few operations on powersets: union, member, intersect, diff as discussed in [Jayaraman & Plaisted 1987]. Some of these operations may, however, be used to define functions which are not continuous. It is up to the user of the metalanguage to prove that the functions are continuous and that it is safe to use the functions for fixpoint iteration.

In the discussion of time complexity we mentioned a domain $\mathbb{N}_{\perp}^{\infty}$ of natural numbers ordered with the usual integer ordering and ∞ as top element. This domain

has not finite height but we may deduce from the use of the domain that it will only use numbers up to a certain limit. If this is not the case then infinity ∞ will be implemented by non-termination. This means that the fixpoint iteration will terminate for all finite numbers but not for the limit point. By this implementation we have achieved what is similar to the usual notation of computability but we can not guarantee decidability (termination). The domain can specified as

domain integer with bottom 0

as the fixpoint operation must know which element to use as bottom element. The equality operation is then assumed to be the same as for **integer**.

Using the lazy fixpoint iteration technique we can construct fixpoint operator for cpo's with type

$$
\begin{aligned}
\tau_{\mathbf{d}} ::= {} & \mathbf{power}\ \ \tau_{\mathbf{f}} \\
\mid {} & \mathbf{domain}\ \tau_{\mathbf{f}}\ \mathbf{with\ bottom\ a_k} \\
\mid {} & \tau_{\mathbf{f}} \rightarrow \tau_{\mathbf{d}} \\
\mid {} & \tau_{\mathbf{d}} \times \tau_{\mathbf{d}} \\
\mid {} & \tau_{\mathbf{d}} + \tau_{\mathbf{d}}
\end{aligned}
$$

and $\tau_{\mathbf{f}}$ is type of a finite domains.

The finite domains are the finite subsets of domains with types given below:

$$
\begin{aligned}
\tau_{\mathbf{f}} ::= {} & \{\mathbf{a}_1, \ldots, \mathbf{a_n}\} \\
\mid {} & \mathbf{integer}\ \ \mid \mathbf{bool}\ \ \mid \mathbf{string} \\
\mid {} & \tau_{\mathbf{f}} \times \tau_{\mathbf{f}} \\
\mid {} & \tau_{\mathbf{f}} + \tau_{\mathbf{f}} \\
\mid {} & \mathbf{rec\ X}.\tau_{\mathbf{f}} \\
\mid {} & \mathbf{X}
\end{aligned}
$$

We can construct fixpoint operators for domains with these types but results will only be defined if the actual domains have finite height and computations only require finite non-bottom values.

4.5.3 Combination

These two rather different approaches to construct fixpoints can now be combined. We can use the lazy fixpoint iteration strategy for finite height domains and **letrec**-expressions for higher-order functions which do not contain domains of finite height. The fixpoint operation will in both cases be simple operators which can be applied and the results evaluated by need.

Interestingly, the two classes of cpo's are not disjoint. For the cpo $\mathbb{N} \rightarrow \mathbb{N}_\perp$ we can use both techniques. This corresponds to the choice between a standard semantics and the minimal function graph interpretation described in section 3.1.

4.5.4 Metalanguage

The metalanguage has been implemented as part of the system described in chapter 7. We will here briefly describe the implementation of the metalanguage and its use as a programming language.

The language is essentially a syntactic sugared version of the language

$$\begin{array}{rcl} \langle \mathbf{exp} \rangle & ::= & \mathbf{lambda}\ \langle \mathbf{name} \rangle \,.\, \langle \mathbf{exp} \rangle \\ & | & (\langle \mathbf{exp} \rangle\ \langle \mathbf{exp} \rangle) \\ & | & \langle \mathbf{name} \rangle \\ & | & \langle \mathbf{number} \rangle \\ & | & \langle \mathbf{string} \rangle \\ & | & (\langle \mathbf{exp} \rangle : \langle \mathbf{type} \rangle) \end{array}$$

where an ML-style typechecker is used to deduce mono-types for all expressions.

The syntax is extended with expressions of the form

$$\begin{array}{rcl} \langle \mathbf{exp} \rangle & ::= & \mathbf{if}\ \langle \mathbf{exp} \rangle\ \mathbf{then}\ \langle \mathbf{exp} \rangle\ \mathbf{else}\ \langle \mathbf{exp} \rangle \\ & | & \mathbf{let}\ \langle \mathbf{name} \rangle = \langle \mathbf{exp} \rangle\ \mathbf{in}\ \langle \mathbf{exp} \rangle \\ & | & \mathbf{letfix}\ \langle \mathbf{name} \rangle = \langle \mathbf{exp} \rangle\ \mathbf{in}\ \langle \mathbf{exp} \rangle \end{array}$$

and a number of operations can also be written in infix form.

The **letfix** expression is a generalised version of the **letrec** expression where

$$\mathbf{letfix}\ \mathbf{x} = \mathbf{F}(\mathbf{x})\ \mathbf{in}\ \mathbf{E}$$

is translated into

$$\mathbf{let}\ \mathbf{x} = \mathbf{F}(\mathsf{fix}(\lambda\, \mathbf{x}.\mathbf{F}(\mathbf{X})))\ \mathbf{in}\ \mathbf{E}$$

such that the fixpoint iteration is only performed if both **F** and **E** depend on **x**.

A number of operations in the language are generic and specialised versions are generated when the metalanguage is translated into the implementation language. Most noticeably there is the function fix which will be constructed as the lazy fixpoint operation if the type allows. The function requires the generic operations eq and bottom to be defined for this type. If the type is higher-order without (non-flat) finite height domains a usual **letrec** expression is generated. If neither case is satisfied the fixpoint cannot be generated within the restrictions of this version of the metalanguage. Notice, however, that the type requirement is only imposed on the fixpoint operator and not on other expressions.

As an example of the metalanguage and its implementation let us consider a few expressions and their evaluation. The fixpoint equation of sets of variable names in section 4.1 can in this language be expressed as

```
letfix D =
  union(singleton("x"),union(
     if member("x",D) then union(singleton("z"),singleton("y"))
                      else singleton("y"),
     if member("y",D) and member("x",D) then singleton("v")
                      else bottom
  )) in D ;
```

The evaluation will then perform the fixpoint iteration and the result is

```
  { "x" , "z" , "y" , "v" }
```

The Fibonacci function can be defined as

```
letfix fib =
   lambda x . if x < 3 then 1 else fib(x-2) + fib(x-1)
in (fib(3)) * (fib(4))  ;
```

The evaluation will give rise to the following calls to take:

evaluate fib(3) :

take$(1, \ldots)$	find fixpoint for 1
take$(2, \ldots)$	find fixpoint for 2

evaluate fib(4) :

take$(2, \ldots)$	value has been found
take$(3, \ldots)$	find fixpoint for 3

All calls to take will evaluate the result immediately, as the evaluation at each call will only depend on values that have already been evaluated.

4.5.5 Implementation

The evaluation of expressions in the metalanguage is performed in four stages:

- Typechecking: all expressions, including generic operations, are assigned types.
- Generic operations: Specialised versions of the generic operations are generated for each type used in the program.
- Translation: The program is translated into the implementation language
- Interpretation: The implementation language is interpreted by a C program.

Restrictions The main restriction with this metalanguage is that we have not solved the problem of higher-order functions over (non-flat) finite height domains. In a standard semantics we may describe programs with the continuation of the type:

$$(\mathbb{A} \rightarrow \mathbf{V}_\perp) \rightarrow \mathbf{V}_\perp$$

Values may in the abstract semantics be modelled by some abstract domain $\mathbf{D}$ and continuation may in this example be described by a domain of type

$$(\mathbb{A} \rightarrow \mathbf{D}) \rightarrow \mathbf{D}$$

This, however, is not implementable with the lazy fixpoint iteration technique but we can implement domains of type

$$\mathsf{list}(\mathbb{A} \times \mathbf{D}) \rightarrow \mathbf{D}$$

where $\mathsf{list}(\mathbb{A} \times \mathbf{D})$ is the domain of lists of values from $\mathbb{A} \times \mathbf{D}$.

It should be possible to implement fixpoint iterators for type $(\mathbf{A} \rightarrow \mathbf{B}) \rightarrow \mathbf{C}$ in a similar fashion to the lazy fixpoint iteration technique but this possibility will not be explored further here.

4.5.6 Conclusion

Denotational semantics can be considered as a programming language if the meaning of programs are expressed in a metalanguage of continuous functions which are λ-definable. In abstract interpretation the standard semantics will often be expressed as a denotational semantics. The abstract semantics, on the other hand, is normally required to be computable in a stronger sense: if it is to be used in a compiler it must be guaranteed to terminate for all programs.

The metalanguage described in this chapter combines these two uses of fixpoint semantics. Both the standard and abstract semantics may be expressible in the metalanguage but proof of termination is now a separate proof obligation. There are, however, examples where we are not expecting to prove termination: the automatic complexity analysis in [Rosendahl 1989] is not guaranteed to terminate as this would be equivalent to solving the halting problem. Except for this, it follows the normal scheme of an abstract interpretation and we would therefore consider termination to be a separate criterion which is not imposed on all interpretations.

The aim here has been generality: we want to find methods that can generate fixpoint iterators for as large a class of functions as possible. There may, however, for certain classes of domains be better ways to perform fixpoint iteration. The frontier algorithm may be more efficient for boolean function as results are only evaluated if they cannot be deduced from earlier found results. For such domains it may be attractive to combine the methods and use the frontiers as a representation of the “previously found fixpoints”.

Chapter 5

Attribute Grammars as Semantic Descriptions

Attribute grammars were introduced in [Knuth 1968] as a convenient way to describe syntax-directed compilations. They were defined as a generalisation of the syntax-directed translation schemes used in some early Algol compilers [Irons 1961].

Attribute grammars provide a powerful implementation language with a substantial collection of efficient evaluators. This has also been used as a semantic language and we will compare some of the notations used for specifying attribute grammars.

5.1 Attribute Grammars

An attribute grammar is a context-free grammar extended with a set of names called *attributes* to each nonterminal and a number of assignment statements per production. An *assignment statement* defines the value of an attribute belonging to a nonterminal in the production. The value may depend on values of other attributes in the production.

Attribute evaluation consists of finding a *consistent set of attribute values* for each node of a parse tree. The attribute values are said to be consistent iff the values of attributes to the node and to other nodes will not change by repeatedly assigning new values, using the assignment statements connected with the production.

An attribute grammar is said to be *circular* if there exists a parse tree where an attribute value is defined (directly or indirectly) from its own value.

An attribute is said to be *synthesised* iff it is only assigned values when the nonterminal occurs on the left hand side of a production. An attribute is called *inherited* if it is only assigned values when the nonterminal occurs on the right hand side.

An attribute grammar is described as *well-formed* if all attributes are either synthesised or inherited and there are assignment statement for each production,

defining all synthesised attributes to the nonterminal on the left hand side and all inherited attributes to nonterminals on the right hand side.

In the original definition [Knuth 1968], only well-formed, non-circular attribute grammars were assigned meanings and under these conditions there is exactly one consistent set of attribute values for a parse tree.

5.1.1 Notation

The usual notation for attribute grammars is somewhat verbose. The attribute definitions are listed after the productions. An attribute instance is represented as a nonterminal followed by the attribute name. The same nonterminal may appear more than once in a production and subscripts to the nonterminals are used to distinguish between attribute instances in these cases.

Three different notations for attribute grammars will be described below with an example. The attribute grammar in the example gives meaning to binary notation.

$$
\begin{array}{ll}
\mathbf{N} \to \mathbf{L} & \mathbf{N}.\mathbf{v} = \mathbf{L}.\mathbf{v} \\
 & \mathbf{L}.\mathbf{s} = 0 \\
\mathbf{L}_1 \to \mathbf{L}_2\mathbf{B} & \mathbf{L}_1.\mathbf{v} = \mathbf{L}_2.\mathbf{v} + \mathbf{B}.\mathbf{v} \\
 & \mathbf{B}.\mathbf{s} = \mathbf{L}_1.\mathbf{s} \\
 & \mathbf{L}_2.\mathbf{s} = \mathbf{L}_1.\mathbf{s} + 1 \\
\mathbf{L} \to \mathbf{B} & \mathbf{L}.\mathbf{v} = \mathbf{B}.\mathbf{v} \\
 & \mathbf{B}.\mathbf{s} = \mathbf{L}.\mathbf{s} \\
\mathbf{B} \to 0 & \mathbf{B}.\mathbf{v} = 0 \\
\mathbf{B} \to 1 & \mathbf{B}.\mathbf{v} = 2^{\mathbf{B}.\mathbf{s}}
\end{array}
$$

The nonterminals **L** and **B** both have a synthesised attribute **v** and an inherited attribute **s**. **N** has a synthesised attribute **v**. The attribute **s** marks the position in the word of the given subtree and the attribute **v** contains the value of the bit pattern in a subtree.

It is a problem with attribute grammars that they often contain many trivial attribute definitions of one attribute being equal to another. In the grammar above there are four of these so-called *copy rules*.

5.1.2 Extended attribute grammars

In extended attribute grammars [Watt & Madsen 1983] there is a syntactic distinction between inherited and synthesised attributes. Inherited attributes and defining expressions are preceded by a down arrow and the synthesised attribute are preceded

by an up arrow.

$$\begin{array}{ll}
\langle \mathbf{N} \uparrow \mathbf{v} \rangle & ::= \langle \mathbf{L} \downarrow 0 \uparrow \mathbf{v} \rangle \\
\langle \mathbf{N} \downarrow \mathbf{s} \uparrow \mathbf{v}1 + \mathbf{v}2 \rangle & ::= \langle \mathbf{L} \downarrow \mathbf{s} + 1 \uparrow \mathbf{v}1 \rangle \, \langle \mathbf{B} \downarrow \mathbf{s} \uparrow \mathbf{v}2 \rangle \\
\langle \mathbf{N} \downarrow \mathbf{s} \uparrow \mathbf{v} \rangle & ::= \langle \mathbf{B} \downarrow \mathbf{s} \uparrow \mathbf{v} \rangle \\
\langle \mathbf{B} \downarrow \mathbf{s} \uparrow 0 \rangle & ::= 0 \\
\langle \mathbf{B} \downarrow \mathbf{s} \uparrow 2^{\mathbf{s}} \rangle & ::= 1
\end{array}$$

The positions for inherited attributes on the left hand side and for synthesised attributes on the right hand side are called *defining positions*. The rest are called *applied positions*. Identifiers occurring in defining positions name the attribute and the applied positions contain expressions to evaluate the attribute value.

The notation provide methods to express some semantic controls directly. It is possible to use constants in defining positions and the intention is then to require that the attribute has exactly that value. If the attribute is assigned a different value during attribute evaluation it is an error which can be reported in a similar fashion to syntactic errors.

5.1.3 Attributed production

We will use a notation similar to extended attribute grammars. It is simplified to make its semantics easier to define. In this notation the defining expressions for attribute values are given after the nonterminals. We will here impose a restriction such that each nonterminal only can have one inherited and one synthesised attribute. We will later relax that by allowing attributes to belong to cartesian products.

An attribute grammar in this notation takes the form of a list of *attributed productions*.

$$\mathbf{p_j} : \ \mathbf{q}_0\{\mathbf{e}_0\} ::= \mathbf{q}_1\{\mathbf{e}_1\} \cdots \mathbf{q_n}\{\mathbf{e_n}\}$$

where

$$\mathbf{p_j} : \ \mathbf{q}_0 ::= \mathbf{q}_1 \cdots \mathbf{q_n}$$

is a production in the underlying grammar and $\mathbf{e}_0, \ldots, \mathbf{e_n}$ are expressions. The expression on the left hand side defines the synthesised attribute of the nonterminal on the left hand side and the expressions on the right hand side define the inherited attributes of the nonterminals on the right hand side. The inherited attribute of the nonterminal on the left hand side can be referenced in the expressions as \$0 and the synthesised attribute to the $\mathbf{i}^{\mathrm{th}}$ on the right hand side can be referenced as \$$\mathbf{i}$. The

production symbol ($\mathbf{p_j}$) will not be included in examples when it can be deduced from the production itself.

$$
\begin{array}{ll}
\mathbf{N}\{\$1\} & ::= \mathbf{L}\{0\} \\
\mathbf{L}\{\$1 + \$2\} & ::= \mathbf{L}\{\$0 + 1\}\mathbf{B}\{\$0\} \\
\mathbf{L}\{\$1\} & ::= \mathbf{B}\{\$0\} \\
\mathbf{B}\{0\} & ::= 0 \\
\mathbf{B}\{2^{\$0}\} & ::= 1
\end{array}
$$

Comparison The three different notations described above are related by the following semantic identities.

$$\mathbf{n}_0\{\mathbf{e}_0\} ::= \mathbf{n}_1\{\mathbf{e}_1\} \cdots \mathbf{n_k}\{\mathbf{e_k}\}$$

$$\Updownarrow$$

$$\langle \mathbf{n}_0 \downarrow \$0 \uparrow \mathbf{e}_0 \rangle ::= \langle \mathbf{n}_1 \downarrow \mathbf{e}_1 \uparrow \$1 \rangle \cdots \langle \mathbf{n_k} \downarrow \mathbf{e_k} \uparrow \$\mathbf{k} \rangle$$

$$\Updownarrow$$

let

$\$0 = \mathbf{n}_0.\mathbf{inh}$

$\$1 = \mathbf{n}_1.\mathbf{syn}$

$\$\mathbf{k} = \mathbf{n_k}.\mathbf{syn}$

in

$\mathbf{n}_0.\mathbf{syn} = \mathbf{e}_0$

$\mathbf{n}_1.\mathbf{inh} = \mathbf{e}_1$

$\vdots$

$\mathbf{n_k}.\mathbf{inh} = \mathbf{e_k}$

5.1.4 Semantics

The meaning of an attributed production is a mapping of a parsetree and the value of the inherited attribute to the synthesised attribute. The meaning of an attribute grammar is an environment of such maps. The semantics follows the usual pattern for recursion equation systems.

The description differs from [Chirica & Martin 1979] in that the semantics is given directly as an interpretation of the attribute grammar rather than as rules to construct a recursion equation system which defines the meaning.

Without loss of generality we may assume that all productions either have a right hand side of **n** nonterminals or a right hand side of one terminal symbol.

Syntactic categories

$$\begin{array}{ll}\mathbf{AG} & : \mathbf{ap}_1 \cdots \mathbf{ap_k} \\ \mathbf{AP} & : \mathbf{q}_0\{\mathbf{e}_0\} ::= \mathbf{q}_1\{\mathbf{e}_1\} \cdots \mathbf{q_n}\{\mathbf{e_n}\} \\ \mathbf{Exp} & : \ \mathbf{expressions}\end{array}$$

Semantic functions

$$\begin{array}{ll}\mathsf{M} & : \mathbf{AG} \to \mathbf{Env} \\ \mathsf{P} & : \mathbf{AP} \to \mathbf{Env} \to \mathbf{T}_\Sigma \to \mathbf{D} \to \mathbf{D} \\ \mathsf{E} & : \mathbf{Exp} \to \mathbf{D^{n+1}} \to \mathbf{D}\end{array}$$

Semantic domains

$$\begin{array}{ll}\mathbf{Env} & : (\mathcal{T}_\Sigma \to \mathbf{D} \to \mathbf{D})^{\mathbf{k}} \\ \mathbf{D} & : \mathbf{attribute\ values} \\ \mathcal{T}_\Sigma & : \mathbf{parse\ trees}\end{array}$$

Semantic rules

$$\begin{array}{l}\mathsf{M}[\![\mathbf{ap}_1 \ldots \mathbf{ap_n}]\!] = \\ \quad \mathsf{fix}(\lambda\,\rho.\,\langle \mathsf{P}[\![\mathbf{ap}_1]\!]\rho, \ldots, \mathsf{P}[\![\mathbf{ap_k}]\!]\rho\rangle) \\ \mathsf{P}[\![\mathbf{n}_0\{\mathbf{e}_0\} ::= \mathbf{n}_1\{\mathbf{e}_1\} \cdots \mathbf{n_k}\{\mathbf{e_k}\}]\!]\rho\ \mathbf{t}\ \mathbf{inh} = \\ \quad \mathbf{let}\ \langle \mathbf{p}, \mathbf{w}_1, \ldots, \mathbf{w_n}\rangle = \mathbf{t}\ \mathbf{and} \\ \qquad \langle \mathbf{p}_{\mathbf{i}_1}, \ldots\rangle = \mathbf{w}_1\ \mathbf{and} \\ \qquad \vdots \\ \qquad \langle \mathbf{p}_{\mathbf{i_n}}, \ldots\rangle = \mathbf{w_n}\ \mathbf{in} \\ \quad \mathsf{E}[\![\mathbf{e}_0]\!](\mathsf{fix}\,\lambda\,\xi.\langle \mathbf{inh},\ (\rho{\downarrow}\mathbf{i}_1)\mathbf{w}_1)(\mathsf{E}[\![\mathbf{e}_1]\!]\xi), \ldots, \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad (\rho{\downarrow}\mathbf{i_n})\mathbf{w_n})(\mathsf{E}[\![\mathbf{e_n}]\!]\xi)\rangle) \\ \mathsf{P}[\![\mathbf{n}_0\{\mathbf{e}_0\} ::= \mathbf{n}_1]\!]\rho\ \mathbf{t}\ \mathbf{inh} = \mathsf{E}[\![\mathbf{e}_0]\!](\langle \mathbf{inh}, \bot, \ldots, \bot\rangle)\end{array}$$

$$\begin{array}{l}\mathsf{E}[\![\$\mathbf{i}]\!]\xi = \xi{\downarrow}\mathbf{i} \\ \mathsf{E}[\![\mathbf{c}]\!]\xi = \mathbf{c} \\ \mathsf{E}[\![\mathbf{e}_1 + \mathbf{e}_2]\!]\xi = \mathsf{E}[\![\mathbf{e}_1]\!]\xi + \mathsf{E}[\![\mathbf{e}_2]\!]\xi \\ \vdots\end{array}$$

The semantic function $\mathsf{P}[\![\mathbf{ap}]\!]$ takes a production environment $\rho \in \mathbf{Env}$ a tree $\mathbf{t} \in \mathcal{T}_\Sigma$ and an inherited value $\mathbf{inh} \in \mathbf{D}$ and returns the value of the synthesised attribute.

The outer level provides the connection between nonterminals and their productions. The inner level gives the connection between attributes in a production. We

do not need to introduce dependency graphs to describe these connections. The semantics can also be used for conditional or circular attribute grammars.

The allowed expressions and the domain of attribute values will not be fixed here. Different choices are possible and will impose different restrictions on the attribute grammars. It is normally possible and desirable to have more than one synthesised and inherited attribute to each nonterminal. This can be obtained if the domain of attribute values (**D**) is a cartesian product of cpo's. The definition in [Chirica & Martin 1979] requires attribute values to belong to flat cpo's. Both definitions are more general than the original definition in [Knuth 1968] since it may give non-bottom attribute values to some parse trees with circular attribute dependencies.

5.1.5 Evaluation

The semantics gives an immediate implementation strategy for attribute grammars. The fixpoints can be computed by **letrec**-expressions in a lazy functional programming language if all base domains are flat cpo's. This allows evaluation of conditional attribute grammars. We may also use the lazy fixpoint iteration technique from chapter 4 to evaluate circular attribute grammars with attribute values in domains of finite height.

5.2 Background

An attribute grammar is a way of assigning a meaning to a string in a context-free language by attributing values to the symbols in a derivation tree for that string. It consists of a context-free grammar augmented with semantic rules for attribute definition. Each nonterminal will have two attributes: one synthesised and one inherited, and each semantic rule will specify values for each occurrence of attributes to nonterminals in the derivation tree. Attribute values are defined according to the semantic rules from other attribute values. Synthesised attributes are defined from the descendants of the corresponding nonterminal symbol and inherited ones from the ancestors of the nonterminal symbol.

Attribute grammars can be seen as extensions of the code-producing actions normally used in LALR(1) parsers (*eg.* YACC: [Johnson 1975]) due to the fact that the context-sensitive information can be propagated to substructures. The context-sensitive part of the code production is normally done by *back-patching*—inserting the right jump addresses and memory locations in already produced code (see [Aho, Sethi & Ullman 1986]) or by using global variables. The code production in YACC-like parsers will normally be described as side-effects to parsing and the resulting code is stored somewhere in the memory. An attribute grammar requires that the semantic rules cannot have side-effects so context-sensitive information can only be propagated via inherited attributes. The parser generator YACC provides facilities

for handling synthesised attributes which are evaluated bottom-up but since the semantic rules are allowed to have side-effects the evaluation depends on parsing being left to right.

Most work in this area has been concerned with developing efficient attribute evaluators, and theoretical work has been concerned with defining subclasses where either checking for well-formedness or evaluation is done more easily. A recent biography and survey [Deransart, Jourdan & Lorho 1988] traces about 600 references on attribute grammars.

5.2.1 Semantics of attribute grammars

The semantics of attribute grammars has a much smaller literature. In the first description ([Knuth 1968]) attributes took values from (possibly infinite) sets. An initial algebraic semantics of attribute grammars with only synthesised attributes is given in [Goguen et al 1977], and this is extended to incorporate inherited attributes and cycles in [Chirica & Martin 1979]. The latter paper uses a fixpoint semantics with *lazy* evaluation of attributes and as such it is an extension of the original definition. [Mayoh 1981] shows how an attribute grammar can be reformulated into an equivalent denotational semantics and that a noncircular attribute grammar can be expressed without using fixed points. The transformation of a denotational semantics to an attribute grammar is considered in [Madsen 1980] where denotations are used as attribute values.

An interesting work on extending the scope of attribute grammars is [Gallier 1984] where attribute definitions (semantic rules) are allowed to contain conditional expressions. The extension follows naturally from the fixpoint semantics given in [Chirica & Martin 1979] as the semantics is well-defined even if the defining expressions for attributes are not strict. Such extensions seem to be important in the current work, as abstract interpretation uses a wider class of mathematical objects than is usual in attribute grammars.

In the original attribute grammar definition, attributes belonged to a set and the attribute relations were required not be circular. This corresponds to the remark in [Nielson & Nielson 1986] that attribute definitions must have the form of a mapping of run-time code to run-time code. They claim that attribute definitions must have type

$$(\mathbf{rt} \underline{\rightarrow} \mathbf{rt}) \times \cdots \times (\mathbf{rt} \underline{\rightarrow} \mathbf{rt}) \rightarrow (\mathbf{rt} \underline{\rightarrow} \mathbf{rt}) \times \cdots \times (\mathbf{rt} \underline{\rightarrow} \mathbf{rt})$$

In other words: attribute values are pieces of code (run-time functions) and during compile-time (attribute evaluation time), an attribute definition takes a number of run-time functions as synthesised and inherited attributes and produces a number of run-time functions as a result.

Correctness The correctness of attribute grammars has been considered by [Deransart 1983] and [Courcelle & Deransart 1988] (see also [Deransart & Maluszynski

1985]). An attribute grammar can be proved correct with respect to a specification by showing that certain assertions will be true for the attributes. Instead of starting with an attribute grammar, the first step is an *attributed scheme* where relations between attributes are described by uninterpreted function symbols. An attribute grammar is then an interpretation of these function symbols over a specified domain. Correctness is shown with a special interpretation, where the attributes are assertions and attribute relations are proof schemes. This is discussed further in section 5.3.

5.2.2 Attribute grammars as semantics

Attribute grammars can be used to define the translation or evaluation of a language. It has also been used to define the semantics of a language without actual evaluation in mind. In [Knuth 1971] a language is given a meaning defined by a semantics in an extended form for attribute grammars. [Paulson 1982] uses so-called semantic grammars to describe meanings of programs in a language and this description is then transformed into a compiler for the language.

Data-flow analysis and attribute grammars Data-flow analysis methods and global optimisation techniques can be used in optimising compilers to improve the performance of programs. A comprehensive bibliography and description of these methods can be found in [Aho, Sethi & Ullman 1986].

A few works have expressed data flow analysis as attribute grammars. In [Wilhelm 1981] constant propagation for a small imperative language was expressed in the language of the MUG2 compiler generator. In [Babich & Jazayeri 1978] live-variable analysis is expressed in the general framework of attribute grammars. However, this is not a real attribute grammar as attribute rules are circular and the attribute rule for the "goto" statement is specified in a non-standard way. This "method of attributes" gives a conceptually simple algorithm for a well-known problem and the underlying system takes care of the main difficulty in most data flow analysis algorithms: how to combine local information into global information.

Neither [Babich & Jazayeri 1978] nor [Wilhelm 1981] use the framework of abstract interpretation to prove the analyses correct.

5.2.3 Attribute evaluation

Attribute evaluators have an extensive literature (see [Deransart, Jourdan & Lorho 1988]) and there are many attribute evaluation systems. [Gallier, Manion & McEnerney 1985] and [Reps & Teitelbaum 1984] are both systems that extend the original definition of attribute grammars. The former describes an efficient depth-first attribute evaluator with lazy evaluation of synthesised attributes, and it seems that an evaluator of this type will cover a large class of abstract interpretations.

Incremental evaluation The latter paper [Reps & Teitelbaum 1984] describes a system designed to generate editors, with attributes evaluated incrementally. The highly efficient evaluator in [Reps & Teitelbaum 1984] will presumably be sufficient for a certain class of abstract interpretations, but restrictions on the attribute grammars seem to prohibit its application in the current work. Further work is needed before incremental attribute evaluation and symbolic composition of attribute grammars can be used for a sufficiently large class of attribute grammars for this project.

Attribute coupled grammars The synthesizer generator [Reps & Teitelbaum 1984] uses a uniform treatment of syntactic and semantic domains. This means that the result of an attribute evaluation (the synthesised attribute for the root symbol) will follow a context-free grammar that can itself be extended to an attribute grammar. This idea is extended further in *attribute coupled grammars* [Ganzinger & Giegerich 1984] with rules for symbolic composition for a restricted class of such attribute grammars. The composition of attribute grammars is similar to the idea in the PSI project ([Nielson & Nielson 1988]) with two levels of interpretation of a program. The meaning is expressed as the composition of these two levels.

5.2.4 Compilers

Abstract interpretation is related to compiler generator systems as they have a common aim: to make it possible to execute the programs according to a specified semantics.

A compiler generator will normally take a semantic specification in a certain meta-language (normally typed lambda calculus) and translate it into a code producing specification. The code production can be done in two steps: first translate the program to a lambda-expression and then translate this expression to code for an abstract machine or real machine ([Paulson 1982] and [Mosses 1983]).

The PSI project uses a slightly different approach in that the semantics is specified in a metalanguage which itself can be given different semantics. One interpretation will be a standard meaning and another will be a code generation interpretation. This two-level specification gives code as the *meaning* of programs and a compilation is then the result of an interpretation of programs under this non-standard semantics.

Other approaches use an operational semantics and translate the specification to code production actions. The Mix project shows this clearly with the compiler generator as a program that transforms an interpreter to a compiler by partial evaluation ([Jones, Sestoft & Søndergaard 1985]).

A compiler generator can be seen as a definition of a very high level language specialised to write compilers in. A language like Pascal was in its design well-suited for compiler writing but we no longer call a Pascal compiler a compiler generator. The phrase "automatic programming" has changed in the same way from programming

using a programming language instead of machine code so it now covers programming in a more "intelligent" environment. There are a few recently designed languages specialised for the writing of interpreters. The language Navel ([Michaelson 1986]) incorporates a parsing/pattern matching interface into a functional programming language and claims it is easy to translate a denotational semantics to an interpreter for the specified language.

Finally, attribute grammars have been used in a number of compiler writing systems. References to systems based on attribute grammars can be found in [Engelfriet 1984] and [Deransart, Jourdan & Lorho 1988]. Besides [Reps & Teitelbaum 1984] and [Gallier, Manion & McEnerney 1985] the Helsinki Language Processor (HLP) (see [Aho, Sethi & Ullman 1986]) has been used for implementing a number of compilers.

5.3 Logical attribute grammars

Many questions about an attribute grammar can be solved without full knowledge of the attribute grammar. All what is needed are the data dependencies: which attributes are needed to evaluate which attributes. To make this more precise [Deransart 1983] introduces an *attributed scheme*.

Informally an *attributed scheme* is an attribute grammar where all expressions in the assignment statements are simplified into function calls. An attribute grammar is then an attributed scheme together with definitions of these functions.

The advantage with this description is that the attribute grammar can be proved well-formed from the attribute scheme alone. It is also plausible that an efficient evaluator can be obtained by preprocessing the attribute scheme, but apparently not much work has been done in this area.

Chapter 6

Correctness Proofs of Attribute Grammars

The purpose of this chapter[1] is to show how an attribute grammar can be proved correct using methods from abstract interpretations.

6.1 Domain attribute grammars

When introducing the ideas from abstract interpretation in the correctness proofs of attribute grammars the aim is to specify two attribute grammars over the same grammar and then establish a relationship between attribute values.

In this section we introduce the concepts required for this type of correctness proof and in the rest of the chapter the method is illustrated by an example. We define a very general class of attribute grammars, called *domain attribute grammars*. In a domain attribute grammar the attributes belong to domains (*i.e.* complete partially ordered sets) and the meaning of a circular attribute definition is defined as the least fixpoint. This class of attribute grammars is mainly meant for proofs and specification. It is expected that many data-flow analysis algorithms can actually be expressed in a sub-class of non-circular attribute grammars where efficient attribute evaluators are available.

When relating two attribute grammars it is important to keep the differences as isolated as possible. For this purpose we use an attributed scheme as described below and the proof will be based on fixpoint induction and logical relations.

6.1.1 Domain attribute grammars

Domain attribute grammars are attribute grammars where attribute values belong to domains (cpo's) and where the attribute rules are continuous functions. This is

[1]The central part of this chapter has been published in Lecture Notes in Computer Science, Volume 461 [Rosendahl 1990]

essentially the class of attribute grammars defined by [Chirica & Martin 1979] although they restrict it to flat cpo's.[2] We use a more general definition which allows a standard semantics to be described in the same framework as the abstract interpretation. It is important for our application that both the standard interpretation and the abstract interpretation can be given a meaning as an attribute grammar but it is only important that the abstract interpretation can be evaluated.

As a minor notational restriction the number of attributes per terminal has been limited to one synthesised and one inherited attribute. When attributes belong to domains, the same effect as having several attributes can be achieved by allowing an attribute type to be the cartesian (or lazy) product of domains. In the same way we could have removed inherited attributes and obtained a simple S-attribute grammar ([Deransart, Jourdan & Lorho 1988]) where the attributes are functions of the former inherited attributes to the former synthesised attributes. There seem to be several good reasons to keep the inherited and synthesised attributes separate in the grammar. Most language specifications will only operate with a limited selection of directions of data-flow. [Cousot & Cousot 1977] distinguishes between forward and backward analysis but mentions that combinations hereof are also possible. The use of inherited and synthesised attributes to describe this direction of data-flow seems to make the specification clearer.

There are two requirements to the attribute definitions in a domain attribute grammar. For all nonterminals we must describe the cpo to which each of the attributes must belong and all attribute definitions must be continuous functions in the attribute values.

Even circular attribute grammars based on these principles will be well-defined as the circularity can be resolved by fixpoint induction.

6.1.2 Attributed scheme

The specification of a domain attribute grammar consists of two parts: a scheme and an associated definition. The scheme is essentially an *attributed scheme* ([Deransart 1983]) where some operator symbols in the attribute rules are defined in the definition part. The difference between the standard interpretation and the abstract interpretation are isolated to the definition part so both use the same attribute scheme.

We will here relax the definition of *attributed scheme* compared to the description in section 5.3. An attributed scheme will here be an attributed production (see section 5.1) where the expressions may contain uninterpreted function symbols (and not just a single function symbol). Each function symbol must be given a type which may contain some uninterpreted type names.

An *interpretation* of the scheme consists of an assignment of cpo's to the type

[2] In their words the "..definition .. is more general than the usual Knuthian semantic definition because [it] assigns a meaning even to those trees in which the attribute dependency is circular."

names and function definitions to the uninterpreted function symbol.

The attribute expressions use an ML-like syntax. The expressions are built from constants and attribute variables ($\$\mathbf{i}, \mathbf{i} = 0, \ldots, \mathbf{n}$) with constructors and function applications. We use a few specialised notations for notational convenience:

- $\mathbf{e}_1, \mathbf{e}_2$ denotes the tuple $(\mathbf{e}_1, \mathbf{e}_2)$
- $\mathbf{e}.1$ selects the first element in a tuple and $\mathbf{e}.2$ selects the second element.
- $\mathbf{f}[\mathbf{x} \mapsto \mathbf{v}]$ is a postfix notation for an updated function:

$$\mathbf{f}[\mathbf{x} \mapsto \mathbf{v}] = \lambda\,\mathbf{y}.\ \mathbf{if}\ \mathbf{y} = \mathbf{x}\ \mathbf{then}\ \mathbf{v}\ \mathbf{else}\ \mathbf{f}(\mathbf{y})$$

6.1.3 Fixpoint induction

In the attribute rules we will leave some operator symbols unspecified for later definition. These symbols will be given two different interpretations where the first will define the standard interpretation and the second the abstract semantics.

We will prove that these two interpretations are related in a way that makes the abstract interpretation safe. The proof will be based on inclusive relations.

6.2 A small language

The following little language will be used to illustrate the use of the attribute grammar framework for abstract interpretation. It is a flow-chart language with assignment statements and unrestricted jumps.

The language will be introduced with a context-free grammar and a discussion of its semantics. A formal semantics for the language will be given in the next section.

$$
\begin{array}{lcl}
\langle\mathbf{program}\rangle & ::= & \langle\mathbf{stmt}\rangle \\
\langle\mathbf{stmt}\rangle & ::= & \langle\mathbf{label}\rangle\ \mathbf{:}\ \langle\mathbf{stmt}\rangle \\
 & | & \mathbf{begin}\ \langle\mathbf{stmt}\rangle\ \mathbf{;}\ \langle\mathbf{stmt}\rangle\ \mathbf{end} \\
 & | & \mathbf{goto}\ \langle\mathbf{label}\rangle \\
 & | & \langle\mathbf{variable}\rangle\ \mathbf{:=}\ \langle\mathbf{exp}\rangle \\
 & | & \mathbf{if}\ \langle\mathbf{exp}\rangle\ \mathbf{then}\ \langle\mathbf{stmt}\rangle\ \mathbf{else}\ \langle\mathbf{stmt}\rangle \\
\langle\mathbf{exp}\rangle & ::= & (\langle\mathbf{exp}\rangle + \langle\mathbf{exp}\rangle) \\
 & | & (\langle\mathbf{exp}\rangle - \langle\mathbf{exp}\rangle) \\
 & | & \langle\mathbf{constant}\rangle \\
 & | & \langle\mathbf{variable}\rangle
\end{array}
$$

The syntactic categories ⟨**variable**⟩ and ⟨**label**⟩ both denote identifiers (strings of letters and digits, starting with a letter) and ⟨**constants**⟩ denotes non negative numbers (a string of digits).

The semantics of the various language constructs are generally as expected. The **goto** command will of course normally send control to the command with the same label. If a label occurs more than once the **goto** command will select the first. A **goto** command to an undefined label causes a jump to the end of the program.

The output from the program is the contents of a given variable (say "x") at the end of the execution. This means that a program will end with an implicit **print**(x) command. Another possibility would have been to let the values of all variables be available at the end of the program but this would make a live-variable analysis less interesting.

6.3 Standard interpretation

This section introduces the standard interpretation for the language. In practice the design of the standard interpretation must go hand in hand with the abstract interpretation and the soundness proof. The general idea in this interpretation is to use a backward or continuation style semantics such that the meaning of a statement in some way describes "the rest of the computation". There are two reasons for using a continuation style semantics: it is partly because the presence of a **goto** statement makes it necessary, and partly because it makes the proof of the live variable analysis easier. In the case of the standard interpretation it is a mapping of the environment at the start of the statement to the final answer. For the abstract interpretation it is the set of variables which may be used in this or later statements along possible execution paths. A variable is said to be *live* at a given program point if it may be used in any execution path before it is assigned a new value. The relationship between these two interpretations can informally be stated as: if a variable is not live at a given program point (*i.e.* dead) then the standard interpretation will not change if that variable is made undefined. We will formalise this in section 6.5.

6.3.1 Attributed scheme

The attribute scheme will show the flow of information in the program but will leave the actual interpretation unspecified. At a later stage a number of operator symbols will be given a meaning which will define either the standard or the abstract interpretation.

The scheme defines a continuation style (or backward) semantics (see [Gordon 1979]). The attributes for statements (⟨**stmt**⟩) has two parts where the second part is used to send information from a labelled statement to a **goto** with that label (a label environment). The first part is the actual meaning and the attributes are linked in such a way that the synthesised attribute is connected to the inherited attribute of its predecessor.

Notice that the attribute dependency is circular, as the attribute rule for ⟨**stmt**⟩ in the first production says the inherited attribute is defined from the synthesised

one. This means that the label environment is defined (implicitly) as a fixpoint from the meaning of the labelled statements.

⟨**program**⟩	{$1.2}
= ⟨**stmt**⟩	{**Final**, $1.2}
⟨**stmt**⟩	{$3.1, $3.2[$1 ↦ $3.1]}
= ⟨**name**⟩ **:** ⟨**stmt**⟩	{$0.1, $0.2}
⟨**stmt**⟩	{$2.1, **Join**($2.2, $4.2)}
= **begin** ⟨**stmt**⟩	{$4.1, $0.2}
; ⟨**stmt**⟩	{$0.1, $0.2}
end	
⟨**stmt**⟩	{**Goto**($0.2, $2), **Nil**}
= **goto** ⟨**name**⟩	
⟨**stmt**⟩	{**Update**($0.1, $1, $3), **Nil**}
= ⟨**name**⟩ **:=** ⟨**exp**⟩	
⟨**stmt**⟩	{**If**($2, $4.1, $6.1), **Join**($4.2, $6.2)}
= **if** ⟨**exp**⟩ **then** ⟨**stmt**⟩	{$0.1, $0.2}
else ⟨**stmt**⟩	{$0.1, $0.2}
⟨**exp**⟩	{**Add**($2, $4)}
= **(** ⟨**exp**⟩ **+** ⟨**exp**⟩ **)**	
⟨**exp**⟩	{**Sub**($2, $4)}
= **(** ⟨**exp**⟩ **-** ⟨**exp**⟩ **)**	
⟨**exp**⟩	{**Const**($1)}
= ⟨**number**⟩	
⟨**exp**⟩	{**Var**($1)}
= ⟨**name**⟩	

The actual flow of information between labelled commands and **goto**'s is not described in the above scheme. This flow can be specified independently of the interpretations with these definitions

$$\mathbf{Goto}(\mathbf{E},\mathbf{l}) = \mathbf{if}\ \mathbf{E}(\mathbf{l}) = \mathbf{Undef}\ \mathbf{then}\ \mathbf{Final}\ \mathbf{else}\ \mathbf{E}(\mathbf{l})$$
$$\mathbf{Nil} = \lambda\,\mathbf{l}.\mathbf{Undef}$$
$$\mathbf{Join}(\mathbf{E}_1,\mathbf{E}_2) = \lambda\,\mathbf{l}.\ \mathbf{if}\ \mathbf{E}_1(\mathbf{l}) = \mathbf{Undef}\ \mathbf{then}\ \mathbf{E}_2(\mathbf{l})\ \mathbf{else}\ \mathbf{E}_1(\mathbf{l})$$

where **Undef** is a special symbol to indicate that no labels have been defined.

It is typical for this type of specification that the nitty-gritty details of actions with undefined labels must be made explicit. They can not be left undefined or to convention.

The attributed scheme can be seen as a program which imports a module that defines the uninterpreted symbols. It is possible to type-check the scheme and give types to the operator symbols. Let **M** and **D** be type names. The attributes will then have the following types:

$$\begin{array}{ll} \text{synthesised of } \langle\textbf{stmt}\rangle & : \mathbf{M} \times (\mathbb{A} \to (\mathbf{M} + \mathbb{U})) \\ \text{inherited of } \langle\textbf{stmt}\rangle & : \mathbf{M} \times (\mathbb{A} \to (\mathbf{M} + \mathbb{U})) \\ \text{synthesised of } \langle\textbf{exp}\rangle & : \mathbf{D} \end{array}$$

The types of the defined symbols are

$$\begin{array}{ll} \textbf{Goto} & : (\mathbb{A} \to (\mathbf{M} + \mathbb{U})) \times \mathbb{A} \to \mathbf{M} \\ \textbf{Nil} & : \mathbb{A} \to (\mathbf{M} + \mathbb{U}) \\ \textbf{Join} & : (\mathbb{A} \to (\mathbf{M} + \mathbb{U})) \times (\mathbb{A} \to (\mathbf{M} + \mathbb{U})) \to (\mathbb{A} \to (\mathbf{M} + \mathbb{U})) \\ \textbf{Undef} & : \mathbb{U} \end{array}$$

and the types for the operator symbols are

$$\begin{array}{ll} \textbf{Update} & : \mathbf{M} \times \mathbb{A} \times \mathbf{D} \to \mathbf{M} \\ \textbf{If} & : \mathbf{D} \times \mathbf{M} \times \mathbf{M} \to \mathbf{M} \\ \textbf{Final} & : \mathbf{M} \\ \textbf{Add} & : \mathbf{D} \times \mathbf{D} \to \mathbf{D} \\ \textbf{Sub} & : \mathbf{D} \times \mathbf{D} \to \mathbf{D} \\ \textbf{Const} & : \mathbb{N} \to \mathbf{D} \\ \textbf{Var} & : \mathbb{A} \to \mathbf{D} \end{array}$$

In the interpretations the type variables **M** and **D** can vary their meaning. $\mathbb{U}$ will always be the singleton set $\{\textbf{Undef}\}$, $\mathbb{N}_\perp$ is the domain of integers, and $\mathbb{A}_\perp$ the domain of identifiers.

In both interpretations **M** and **D** will be given the same interpretation. Their intuitive meaning is different—respectively the meaning of “the rest of the program” and the meaning of “the expression”—so we will keep them separate in the interpretations and the proof. It is possible to define an abstract interpretation where these types are bound to different domains.

6.3.2 Standard interpretation

In the standard interpretation the meaning (of type **M**) of a statement will be a mapping of the environment before the statement to the final result. The final result is the value of the variable “**x**” at the end of the program.

For expressions the meaning (of type **D**) is a mapping of the environment to the value of the expression.

The standard interpretation uses the following interpretation of type names:

$$\mathbf{M} : \mathbf{env} \rightarrow \mathbb{Z}_\perp$$
$$\mathbf{D} : \mathbf{env} \rightarrow \mathbb{Z}_\perp$$
$$\mathbf{env} : \mathbb{A} \rightarrow \mathbb{Z}_\perp$$

and the interpretation of operator symbols are

$$\begin{aligned}
&\mathbf{Update}(\mathbf{Cn}, \mathbf{n}, \rho) &&= \lambda\, \mathbf{y}.\mathbf{Cn}(\mathbf{y}[\mathbf{n} \mapsto \rho(\mathbf{y})]) \\
&\mathbf{If}(\rho, \mathbf{Ct}, \mathbf{Cf}) &&= \lambda\, \mathbf{y}.\ \mathbf{if}\ \rho(\mathbf{y})\ \mathbf{then}\ \mathbf{Ct}(\rho)\ \mathbf{else}\ \mathbf{Cf}(\rho) \\
&\mathbf{Final} &&= \lambda\, \mathbf{y}.\rho(\text{“x”}) \\
&\mathbf{Add}(\rho 1, \rho 2) &&= \lambda\, \mathbf{y}.\rho 1(\mathbf{y}) + \rho 2(\mathbf{y}) \\
&\mathbf{Sub}(\rho 1, \rho 2) &&= \lambda\, \mathbf{y}.\rho 1(\mathbf{y}) - \rho 2(\mathbf{y}) \\
&\mathbf{Const}(\mathbf{n}) &&= \lambda\, \rho.\mathbf{n} \\
&\mathbf{Var}(\mathbf{v}) &&= \lambda\, \rho.\rho(\mathbf{v})
\end{aligned}$$

6.4 Live-variable analysis

The live-variable analysis will be specified as an interpretation of the above attributed scheme. The meaning of a statement is the set of live variables at the start of the statement, and the meaning of an expression is the set of variables used in it.

The interpretation of type names is

$$\mathbf{M} : \mathbb{P}(\mathbb{A})$$
$$\mathbf{D} : \mathbb{P}(\mathbb{A})$$

This is the powerset of variable names ordered by set inclusion. The interpretation of operator symbols is

$$\begin{aligned}
&\mathbf{Update}(\mathbf{Cn}, \mathbf{n}, \mathbf{e}) &&= \mathbf{e} \cup (\mathbf{Cn} \backslash \{\mathbf{n}\}) \\
&\mathbf{If}(\mathbf{e}, \mathbf{Ct}, \mathbf{Cf}) &&= \mathbf{e} \cup \mathbf{Ct} \cup \mathbf{Cf} \\
&\mathbf{Final} &&= \{\text{“x”}\} \\
&\mathbf{Add}(\mathbf{e}1, \mathbf{e}2) &&= \mathbf{e}1 \cup \mathbf{e}2 \\
&\mathbf{Sub}(\mathbf{e}1, \mathbf{e}2) &&= \mathbf{e}1 \cup \mathbf{e}2 \\
&\mathbf{Const}(\mathbf{n}) &&= \varnothing \\
&\mathbf{Var}(\mathbf{v}) &&= \{\mathbf{v}\}
\end{aligned}$$

All these functions can easily be seen to be continuous. It is not difficult to define attribute functions that are not continuous as the set difference operation ($\backslash$) is not continuous. In this case it only subtracts a singleton set and that operation is continuous.

6.5 Correctness proof of the abstract interpretation

In the correctness proof we will introduce a relationship between values in the two interpretations of the base types. The index **s** will be used for the standard interpretation and **a** for the abstract interpretation. Thus we will establish a relation $\trianglerighteq_{\mathbf{M}}$ between $\mathbf{M_s}$ and $\mathbf{M_a}$ and a relation $\trianglerighteq_{\mathbf{D}}$ between $\mathbf{D_s}$ and $\mathbf{D_a}$. The relation can be extended to other types in our type structure as described in section 6.1.

The proof is in two parts. In the first part we prove that the two interpretations of the operator symbols can be related. In the second part we prove that the relation is admissible and hence can be extended to fixpoints. From this we conclude that attributes in the two interpretations will be related for all programs.

To prove a relation between two interpretations of the language we must make sure that

- all attribute rules are continuous functions.
- a relation holds between values of the base types (**M** and **D**).
- that the relations relate the interpretations of the operator symbols
- that the relation is strict and inclusive and hence can be extended to composite types and to fixpoints.

6.5.1 Relation

The relation for **M** and **D** is the same and is defined as

$$\mathbf{c_s} \trianglerighteq_{\mathbf{M}} \mathbf{c_a} \Leftrightarrow \forall\, \mathbf{x}.\mathbf{x} \notin \mathbf{c_a} \Rightarrow \forall\, \mathbf{e}.\mathbf{c_s}(\mathbf{e}) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \bot])$$

This can be interpreted as: if a variable is not live then the standard interpretation of the rest of the program will not change if its value is undefined. The interpretation for **D** is the same except "the rest of the program" should be "the whole of the expression".

The relation does not say that a live variable will be used in later statements but only that if a variable is not live, it will not be used.

6.5.2 Local correctness

The correctness proof requires that the relation is proved for the interpretations of the operator symbols. The proof is essentially no more than symbol manipulation. It should be done for all the operator symbols but we will only include the proof for **Update**.

We will prove that

$$\mathbf{Update_s} \trianglerighteq_{(\mathbf{M}\times\mathbb{A}\times\mathbf{D})\to\mathbf{M}} \mathbf{Update_a}$$

or equivalently that

$$\forall\, \mathbf{c_s}, \mathbf{c_a}, \mathbf{v_s}, \mathbf{v_a}, \mathbf{e_s}, \mathbf{e_a} :$$
$$\mathbf{c_s} \trianglerighteq_{\mathbf{M}} \mathbf{c_a}, \mathbf{v_s} = \mathbf{v_a}, \mathbf{e_s} \trianglerighteq \mathbf{e_a} \Rightarrow \mathbf{Update_s}(\mathbf{c_s}, \mathbf{v_s}, \mathbf{e_s}) \trianglerighteq_{\mathbf{M}} \mathbf{Update_a}(\mathbf{c_a}, \mathbf{v_a}, \mathbf{e_a})$$

Let $\mathbf{c_s} \in \mathbf{M_s}, \mathbf{c_a} \in \mathbf{M_a}, \mathbf{v_s} \in \mathbb{A}, \mathbf{v_a} \in \mathbb{A}, \mathbf{e_s} \in \mathbf{D_s}, \mathbf{e_a} \in \mathbf{D_a}$ be any values such that

$$\mathbf{c_s} \trianglerighteq_{\mathbf{M}} \mathbf{c_a}, \mathbf{v_s} = \mathbf{v_a}, \mathbf{e_s} \trianglerighteq \mathbf{e_a}$$

We need to prove that

$$\mathbf{Update_s}(\mathbf{c_s}, \mathbf{v_s}, \mathbf{e_s}) \trianglerighteq_{\mathbf{M}} \mathbf{Update_a}(\mathbf{c_a}, \mathbf{v_a}, \mathbf{e_a})$$

Let $\mathbf{v} = \mathbf{v_a} = \mathbf{v_s}$ and let $\mathbf{x}$ be any variable such that $\mathbf{x} \notin \mathbf{c_a}$ and $\mathbf{e}$ any environment $\mathbf{e} \in \mathbf{env_s}$. The relationship we seek is then

$$\mathbf{c_s}(\mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})]) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \bot][\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e}[\mathbf{x} \mapsto \bot])])$$

using $\mathbf{e_s}(\mathbf{e}) = \mathbf{e_s}(\mathbf{e}[\mathbf{x} \mapsto \bot])$ gives

$$\mathbf{c_s}(\mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})]) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \bot][\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})])$$

if $\mathbf{v} = \mathbf{x}$ then there is no more to prove; else assume $\mathbf{x} \notin \mathbf{c_a}$

$$\mathbf{c_s}(\mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})]) = \mathbf{c_s}(\mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})][\mathbf{x} \mapsto \bot])$$

and with $\mathbf{e}_1 = \mathbf{e}[\mathbf{v} \mapsto \mathbf{e_s}(\mathbf{e})]$ we have

$$\mathbf{c_s}(\mathbf{e}_1) = \mathbf{c_s}(\mathbf{e}_1[\mathbf{x} \mapsto \bot])$$

This is true by the assumption $(\mathbf{c_s} \trianglerighteq_{\mathbf{M}} \mathbf{c_a} \wedge \mathbf{x} \notin \mathbf{c_a})$.

The proof for the other operator symbols is performed in the same fashion.

6.5.3 Fixpoint induction

The final part of the proof is to show that the relation $\trianglerighteq_{\mathbf{M}}$ is admissible. That is to prove that for any directed subset $\mathbf{X} \subseteq \trianglerighteq_{\mathbf{M}}$ that $\bigsqcup \mathbf{X} \subseteq \trianglerighteq_{\mathbf{M}}$. This follows from the fact that $\mathbf{M_a}$ is finite (there is only a finite number of variables in a program) and that $\forall\, \mathbf{e}.\mathbf{c_s}(\mathbf{e}) = \mathbf{c_s}(\mathbf{e}[\mathbf{x} \mapsto \bot])$ is an admissible predicate on $\mathbf{M_s}$. This follows from simple structural argument due to [Manna, Ness & Vuillemin 1972] (see also [Stoy 1977]).

6.6 Implementation of the attribute grammar

The attribute scheme presented in section 6.3 is clearly circular but both the standard semantics and the abstract interpretation can easily be implemented. The circularity in the attributed scheme occurs for the second part of the synthesised attribute to ⟨**stmt**⟩. This part has type $\mathbb{A} \to (\mathbf{M} + \mathbb{U})$. The fixpoint iteration will be performed in different ways in the two interpretations.

In the standard interpretation the recursively defined attribute has type $\mathbb{A} \to (\mathbf{env} \to \mathbb{Z}_\perp)$. This is a functional object and it is therefore just a definition of a recursive function.

For the abstract interpretation the circular definition is for objects of type $\mathbb{A} \to \mathbb{P}(\mathbb{A})$. The evaluation of this type of object will require fixpoint iteration but it is guaranteed to terminate as a program can only contain a finite number of labels and a finite number of variables. This fixpoint iteration can be performed using the lazy fixpoint iteration technique described in chapter 4. For the standard semantics the type of the fixpoint is a higher order function and the implementation uses a **letrec** expression. The abstract semantics, on the other hand, is based on domains of finite height.

Chapter 7

Implementation of Abstract Interpretation using Attribute Grammars

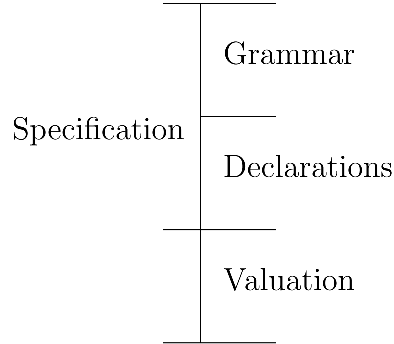
The aim of this chapter is to outline an evaluator system for a class of domain theoretic attribute grammars. The evaluator is based on attributed schemes as described in section 5.3 and 6.3 and the implementation uses the lazy fixpoint iteration technique described in chapter 4.

7.1 Description

The structure of our attribute evaluator system is based on the ideas of Reynolds scheme for polymorphism ([Reynolds 1983] and section 3.3) and attributed productions (section 5.3).

An attribute grammar will be separated into two parts: a *specification* and a *valuation*. The specification corresponds to an attributed scheme [Deransart 1983], a *version* (section 3.3), or a skeleton semantics [Jones & Mycroft 1986]. A specification consists of a context-free grammar with attribute definitions and some declarations and auxiliary functions. The specification may contain some undefined type names and operator symbols.

A valuation assigns domains to type names and contains definitions of remaining operator symbols. These parts may be sketched as:



The specification alone can be analysed as a polymorphically typed definition where we require the type variables named. Further, we can make a list of type variables and operator names which have not yet been given an interpretation. As such the specification is a module parameterised by type names and operators. The valuation must then contain one and only one definition for each type variable and operator name in the specification.

The specification defines a signature with type variables as sorts and uninterpreted operator symbols as operators. The valuation defines an algebra over this signature and describes how to evaluate the synthesised attribute of the root.

The idea is that the valuation can be reused for different languages and specifications should be the same for different interpretations. In this sense the specification defines a metalanguage and the valuation an interpretation of the metalanguage.

7.2 Specification

The syntax of the input language is given below. The implementation and use of the system is described later. The attribute evaluation is based on the metalanguage from chapter 4 and the nonterminals **type** and **expression** are not defined here.

attribute-grammar	::=	*grammar-part valuation-part*
grammar-part	::=	`grammar` *nonterminal*
		`is` *grammar*
		`where` *declaration-part* `end`
declaration-part	::=	*definition declaration-part*
	|	*attribute declaration-part*
	|	ϵ
valuation-part	::=	*definition valuation-part*
	|	ϵ
definition	::=	`type` *name* `=` *type*
	|	`val` *name* `=` *expression*
attribute	::=	`synthesised` *nonterminal-list* `:` *type*
	|	`inherited` *nonterminal-list* `:` *type*

```
nonterminal-list   ::= nonterminal , nonterminal-list
                     | nonterminal
grammar            ::= production grammar
                     | production
production         ::= lhs = symbol-list
lhs                ::= nonterminal
                     | nonterminal < expression >
symbol-list        ::= symbol | symbol-list
                     | symbol
symbol             ::= " string "
                     | name
                     | number
                     | nonterminal
                     | nonterminal < expression >
```

An example of an attribute grammar given in this specification language is to be found later.

Definitions can either assign domains to type names or define global constants (typically of functional types).

Types are built from base types: integers, booleans, and identifiers with type constructors. Expressions are built from basic operations and variables with lambda abstraction, function application, and so on. In attribute definitions a special expression can be used to access attributes to other nonterminals in the production. The expression

```
expression         ::= $ number
```

is taken from YACC and "`$` **n**" ($\mathbf{n} > 0$) is the value of the synthesised attribute to the $\mathbf{n}^{\text{th}}$ symbol on the right hand side of the production. "`$0`" refers to the inherited attribute of the left hand side.

Attributes are defined by placing an expression in curly brackets after the nonterminal. An expression placed after the left hand side nonterminal defines its synthesised attribute. The default value for inherited attributes is the inherited attribute of the left hand side nonterminal. The default value of the synthesised attribute on the left hand side is the first synthesised attribute on the right hand side.

The specification (the declaration and the grammar) is not required to specify types for all type variables or define all operators but with the valuation all expressions must be strongly (mono-) typed. This will enable the system to use specialised fixpoint iterations.

The evaluation order will be lazy to make circular attribute definitions have their intended meaning.

7.3 Proof rules

An abstract interpretation in this system consists of one specification, two valuations of the specification, and relations for all undefined type names in the specification. A proof of correctness will consist of

- Well-definedness of the interpretations. This means that all attribute expressions must be continuous functions in the other attribute values.
- Well-definedness of the relations. All relations must be inclusive.
- Relationship. The relations must be proved for the operator symbols in the valuations.
- Implementability. At least the abstract semantics is expected to be implemented. We must then prove that all fixpoint operators have types which can be implemented in the metalanguage from chapter 4. We may also prove that all fixpoints are over domains of finite height if termination is a requirement.

7.4 Implementation

The input to the system consists of two parts: the grammar part and the valuation. The grammar part is taken as a polymorphic function definition where some operator symbols and type names are left undefined. These names and symbols must be defined in the valuation part.

The system is separated into two parts: From the attribute grammar, we generate an evaluator; and the evaluator takes a source program and computes the synthesised attribute to the root symbol.

The generator performs the following tasks

- The specification is typechecked and we generate a list of undefined type names and operator symbols.
- The valuation is typechecked and each undefined symbol and type name from the specification must have a definition in the valuation.
- All attribute expressions and function definitions are translated into the implementation language (see section 4.5). Specialised versions for generic operations are generated for each type used in the grammar. This translation follows the semantics for attributed productions in section 5.1
- The context-free grammar is extracted and a parser is generated using YACC and LEX.

- Semantic actions to the parser are generated. The actions will at parse time construct expressions in the implementation language which will evaluate attribute values.

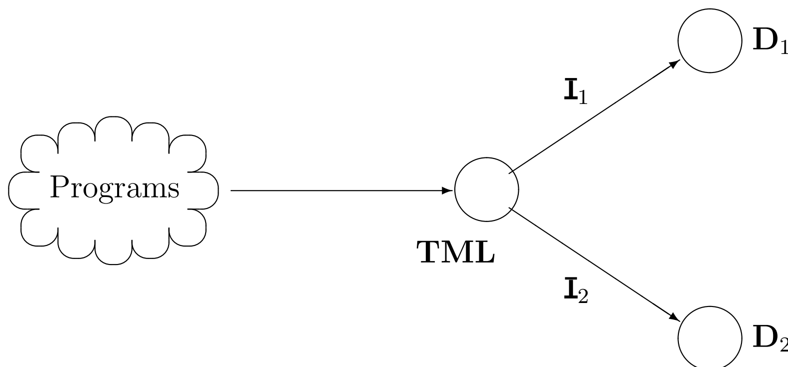
The attribute evaluator works in two phases. In the first phase an expression is constructed from the input. In the second phase this expression is then evaluated.

7.5 Comparison

There are other systems which perform related tasks in program analysis and compiler generation. We will compare two different systems with the present one.

Two-level metalanguage The PSI-system [Nielson 1987] is an implementation of some of the principles behind the project around a denotational abstract interpretation based on a *two-level metalanguage* (TML) [Nielson 1989]. The goal behind this approach is to construct a metalanguage which can be used as an intermediate language for a number of programming languages. From this metalanguage it should be possible to perform a number of different interpretations for program analysis or compilation. This is essentially the old UNCOL dream [Aho, Sethi & Ullman 1986] though it is here founded on denotational semantics which has a proven generality for language description.

The framework can be illustrated as follows



This approach seems superior to the system described here as an interpretation can be used for all languages. It is only necessary to specify the translation into the metalanguage to get access to the collection of analyses. Experience shows, however, that the selection of datatypes in programming languages vary and specialised versions of TML have been defined with different type structures. Furthermore, it is not possible or difficult to define some analyses for the full TML. This means that a whole family of versions of TML have been defined ($\mathbf{TML_m}, \mathbf{TML_s}, \mathbf{TML_b}, \ldots$). Another aspect of this way of making interpretations of a metalanguage is that it can be difficult to see whether the results can be used for program transformations.

The interpretation produces information about the metalanguage but it may not be easy to see the implication for the source program.

Our system is less ambitious than TML in that it is directed towards designing an analysis for a specific language. The separation into a specification and a valuation is so far mainly seen as a convenient way to structure the proof. However, we may understand the system as a method to define versions of TML. It may be possible to reuse a valuation for different specifications so that it effectively defines a metalanguage and the same specification may be used for different analyses. Whether this is possible or attractive is yet an open question as it is difficult to see how much an instrumented semantics or otherwise unusual standard interpretation will affect the specification.

Compiler generators There are several compiler generator systems based on denotational semantics and domain theory. A system which close in approach to this system is reported in [Paulson 1982]. It uses denotational semantics expressed in an attribute grammar formalism and transform the description into a compiler. The similarities are many: it uses an extension to the usual attribute grammar definition called *semantic grammars* which are nearly the same as our domain-theoretic attribute grammars. There are, however, some important differences in the applications.

The semantic grammars is used to describe denotational semantics. This type of description is easier to implement than the typical expressions in abstract interpretation.

The compiler generator is used to construct a compiler from a semantic description. The compiler should will then from a source program produce code for a stack machine. Our system is used to perform interpretations of programs. It may be possible to construct a compiler but one should then use a code producing interpretation [Nielson & Nielson 1988].

7.6 Example

In [Gallier 1984] an example shows an attribute grammar with conditionals for evaluating signed binary numbers with overflow detection. An attribute grammar for evaluating binary numbers was first given in [Knuth 1968] and has since been the standard example in attribute grammars.

The attribute grammar describes valuation of binary numbers. The binary numbers follow the context-free grammar:

$$
\begin{array}{ll}
\mathbf{Z} \rightarrow \mathbf{A}\ \mathbf{L} & \\
\mathbf{A} \rightarrow + & \mid - \\
\mathbf{L} \rightarrow \mathbf{L}\ \mathbf{B} & \mid \mathbf{B} \\
\mathbf{B} \rightarrow 0 & \mid 1
\end{array}
$$

The example in [Gallier 1984] is as follows:

$$
\begin{array}{lll}
\mathbf{p}_1 : \mathbf{Z} \to \mathbf{A}\ \mathbf{L} & \mathbf{Z}.\mathbf{v} = \mathbf{cond}(\mathbf{A}.\mathbf{neg}, \mathbf{f}_4(\mathbf{L}.\mathbf{v}), \mathbf{L}.\mathbf{v}) & \\
 & \mathbf{Z}.\mathbf{over} = \mathbf{L}.\mathbf{over} & \mathbf{L}.\mathbf{s} = \alpha_0 \\
\mathbf{p}_2 : \mathbf{A} \to + & \mathbf{A}.\mathbf{neg} = \mathbf{false} & \\
\mathbf{p}_3 : \mathbf{A} \to - & \mathbf{A}.\mathbf{neg} = \mathbf{true} & \\
\mathbf{p}_4 : \mathbf{L} \to \mathbf{B} & \mathbf{L}.\mathbf{v} = \mathbf{cond}(\mathbf{less}(\mathbf{L}.\mathbf{s}, \mathbf{M}), \mathbf{B}.\mathbf{v}, \mathbf{f}_1) & \\
 & \mathbf{L}.\mathbf{over} = \mathbf{cond}(\mathbf{less}(\mathbf{L}.\mathbf{s}, \mathbf{M}), \mathbf{false}, \mathbf{true}) & \\
 & & \mathbf{B}.\mathbf{s} = \mathbf{L}.\mathbf{s} \\
\mathbf{p}_5 : \mathbf{L}_1 \to \mathbf{L}_2\ \mathbf{B} & \mathbf{L}_1.\mathbf{v} = \mathbf{cond}(\mathbf{L}_1.\mathbf{over}, \mathbf{f}_2(\mathbf{L}_1.\mathbf{v}, \mathbf{B}.\mathbf{v}), \mathbf{f}_1(\mathbf{M})) & \\
 & \mathbf{L}_1.\mathbf{over} = \mathbf{L}_2.\mathbf{over} & \\
 & & \mathbf{L}_2.\mathbf{s} = \mathbf{f}_3(\mathbf{L}_1.\mathbf{s}) \\
 & & \mathbf{B}_2.\mathbf{s} = \mathbf{L}_1.\mathbf{s} \\
\mathbf{p}_6 : \mathbf{B} \to 0 & \mathbf{B}.\mathbf{v} = \alpha_0 & \\
\mathbf{p}_7 : \mathbf{B} \to 1 & \mathbf{B}.\mathbf{v} = \mathbf{f}_1(\mathbf{B}.\mathbf{s}) &
\end{array}
$$

with the additional definitions

$$
\begin{aligned}
\mathbf{f}_1(\mathbf{x}) &= 2^{\mathbf{x}} \\
\mathbf{f}_2(\mathbf{x}, \mathbf{y}) &= \mathbf{x} + \mathbf{y} \\
\mathbf{f}_3(\mathbf{x}) &= \mathbf{x} + 1 \\
\mathbf{f}_4(\mathbf{x}) &= -\mathbf{x} \\
\alpha_0 &= 0 \\
\mathbf{M} &= 32
\end{aligned}
$$

Analysis We will now derive an abstract interpretation of binary numbers that estimates whether it contains an even or odd number of ones ("1"). The interpretation will return a "don't know" value in case of overflow in the number. The interpretation will be made over a value set containing three values : **odd**, **even** and **either**.

To do this in our system we need to define a specification and give two valuations. It is then important to decide which values may vary between valuations and which values should always be given the same meaning.

We may expect the position of binary digit (described by the attribute **s**), the sign of the number (described by the attribute **neg**), and the overflow condition (attribute **over**), are given the same meaning in both interpretations. The value of the number, however, may vary between the interpretations. All definitions that cannot vary between valuations may be defined in the specification leaving only α_1, $\mathbf{f}_1$, $\mathbf{f}_2$, and $\mathbf{f}_4$ in the valuation part.

In TML [Nielson & Nielson 1988], one would specify the type of the attribute v as a run-time type and s as a compile-time type.

Specification Expressed in our system the specification part for the binary numbers may be given as below. We will introduce one minor change in the attribute grammar. The constant α_0 is used in two different meanings in the example: to initialize the position counter (in production $\mathbf{p}_1$) and as the value of the binary number "0" (in $\mathbf{p}_6$). We will use the constant α_1 for the second meaning.

```
grammar Z is
  Z <mksigned($1,$2)> = A L <a0>
  A <false>         = "+"
  A <true>          = "-"
  L <first($0,$1)> = B <$0>
  L <add($0,$1)>    = L <f3($0)> B <$0>
  B <a1>            = "0"
  B <f1($0)>        = "1"
where
  synthesised Z : value * bool ;
  inherited   Z : empty ;
  inherited   L : position ;
  type position = integer ;

  val f3(x) = x + 1 ;
  val a0    = 0 ;
  val M     = 32 ;
  val mksigned(x,(y1,y2)) = if x then (f4(y1),y2) else (y1,y2)
  val first(x,y)          = if x < M then (y,false) else (f1(M),true)
  val add((x1,x2),y)      = if x2 then (x1,x2) else (f2(x1,y),x2)
end
```

The specification part can be seen as a module parameterised with a set (**value**), a constant ($\mathbf{a}_1$), and three operators ($\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_4$). The abstract interpretation will use other values for these parameters.

The specification language requires that all attributes are strongly typed and we have here used the name `value` for the values of binary numbers, `position` for the position of binary digits in numbers, `bool` for boolean values (both the overflow condition and minus-sign condition), and `empty` for unused attribute positions.

Standard interpretation The standard interpretation will be equivalent to the attribute grammar above.

```
type value  = integer ;

val f1(x)   = exp(2,x) ;
val f2(x,y) = x + y ;
val f4(x)   = -x ;
val a1      = 0 ;
```

Abstract interpretation The abstract interpretation can be expressed in the specification language as:

```
type value = {odd,even,either} ;

val f1(x) = if x < M then odd else either ;
val f2(x,y) = if x = either or y = either then either
              else if x = y then even
                            else odd ;
val f4(x) = x ;
val a1 = even ;
```

Proof Assume we have defined functions isodd and iseven which tests whether a number is odd or even. The relation between values in the two standard and abstract semantics is then

$$\mathbf{x} \trianglelefteq_{\mathbf{value}} \mathbf{x}' \quad \stackrel{\text{def}}{\Leftrightarrow} \mathbf{x}' = \mathbf{either} \vee (\mathbf{x}' = \mathbf{odd} \wedge \mathsf{isodd}(\mathbf{x})) \vee (\mathbf{x}' = \mathbf{even} \wedge \mathsf{iseven}(\mathbf{x}))$$

This relation can then be lifted through the type structure and our proof obligation is that the standard interpretation of operator symbols ($\mathbf{I}[\![\cdot]\!]$) is related to the abstract interpretation ($\mathbf{I}'[\![\cdot]\!]$).

$$\begin{array}{ll}
\mathbf{I}[\![\mathbf{a}_1]\!] \trianglelefteq_{\mathbf{value}} \mathbf{I}'[\![\mathbf{a}_1]\!] & \mathbf{or}\ 0 \trianglelefteq_{\mathbf{value}} \mathbf{even} \\
\mathbf{I}[\![\mathbf{f}_1]\!] \trianglelefteq_{\mathbf{position}\to\mathbf{value}} \mathbf{I}'[\![\mathbf{f}_1]\!] & \mathbf{or}\ \forall \mathbf{x} : \mathbf{if}\ \mathbf{x} < \mathbf{M}\ \mathbf{then}\ 2^{\mathbf{x}} \trianglelefteq_{\mathbf{value}} \mathbf{odd} \\
& \qquad\qquad \mathbf{else}\ 2^{\mathbf{x}} \trianglelefteq_{\mathbf{value}} \mathbf{either} \\
\mathbf{I}[\![\mathbf{f}_2]\!] \trianglelefteq_{\mathbf{value}\times\mathbf{value}\to\mathbf{value}} \mathbf{I}'[\![\mathbf{f}_2]\!] & \mathbf{or}\ \forall \mathbf{x}, \mathbf{x}', \mathbf{y}, \mathbf{y}' : \mathbf{x} \trianglelefteq_{\mathbf{value}} \mathbf{x}' \wedge \mathbf{y} \trianglelefteq_{\mathbf{value}} \mathbf{y}' \Rightarrow \\
& \qquad\qquad \mathbf{x} + \mathbf{y} \trianglelefteq_{\mathbf{value}} \mathbf{x}' \oplus \mathbf{y}' \\
& \text{— which can be shown by case analysis} \\
\mathbf{I}[\![\mathbf{f}_4]\!] \trianglelefteq_{\mathbf{value}\to\mathbf{value}} \mathbf{I}'[\![\mathbf{f}_4]\!] & \mathbf{or}\ \forall \mathbf{x} : \mathbf{x} \trianglelefteq_{\mathbf{value}} \mathbf{x}' \Rightarrow -\mathbf{x} \trianglelefteq_{\mathbf{value}} \mathbf{x}'
\end{array}$$

The relation extends to **mksigned**, **first**, and **add** using

$$\mathbf{I}[\![\mathbf{cond}]\!] \trianglelefteq_{\mathbf{bool}\times\mathbf{value}\times\mathbf{value}\to\mathbf{value}} \mathbf{I}'[\![\mathbf{cond}]\!]$$

where

$$\mathbf{cond}(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{if}\ \mathbf{x}\ \mathbf{then}\ \mathbf{y}\ \mathbf{else}\ \mathbf{z}$$

and hence to all attribute definitions [not involving fixpoints].

To prove that the relation $\trianglelefteq$ also holds for the attribute values we need to show that the relation is inclusive:

$$\begin{array}{l}
\trianglelefteq_\sigma \ \text{inclusive} \\
\Updownarrow \\
(\mathbf{x_i}) \in \mathbf{I}[\![\sigma]\!] \wedge (\mathbf{x'_i}) \in \mathbf{I}'[\![\sigma]\!] \wedge (\forall \mathbf{i} : \mathbf{x_i} \trianglelefteq_\sigma \mathbf{x'_i}) \Rightarrow \bigsqcup\{\mathbf{x_i}\} \trianglelefteq_\sigma \bigsqcup\{\mathbf{x'_i}\}
\end{array}$$

This is obviously true for the basic domains ($\mathbf{I}[\![\mathbf{value}]\!]$ and $\mathbf{I}'[\![\mathbf{value}]\!]$) in our example and the logical structure of the relation extends it to all types. As the relation is satisfied between the attribute definitions the relationship will also hold between the fixpoints. □

Chapter 8

Conclusion

The concluding remarks will mainly be in the form of ideas for future directions of research. We have described the development of a system to implement abstract interpretation using attribute grammars. The ideas behind the system can also be used for correctness proofs of attribute grammars. The system is designed as an experimental system for automatic implementation of abstract interpretation. It includes a method to generate fixpoint iterators for a class of cpo's. This technique can also be used separately as a domain theoretic language.

Fixpoint iteration There are some direct shortcomings of the fixpoint iteration technique described in chapter 4. We have not solved the problem of fixpoints for higher-order functions to non-flat base domains. A likely solution may be to consider closure semantics for λ-lifted expressions. We have, however, not pursued this idea.

Another improvement to the method is a better way to find fixpoints of functions which depend on fixpoints. The current method will reevaluate the inner fixpoint at each iteration even though continuity means it can start fixpoint iteration at the last value. We have experimented with methods to "flatten" such expressions into fixpoints for cartesian products. A more comprehensive treatment may require some analysis (by abstract interpretation! evaluation order or strictness analysis) of the metalanguage. We have not examined those ideas any further.

The implementation of the domain theoretic language is used in the attribute evaluator system but it is interesting on its own. The language with **letfix**-expressions is surprisingly powerful and gives a whole new meaning to the phrase "recursive programming".

The attribute evaluator system We have constructed an attribute evaluator system for implementing abstract interpretations. It is at the moment only meant as an experimental system in that the user interface and the speed can be improved.

It is worth considering which notation to use for specifying abstract interpretations. The present system emphasises brevity of specification but more experience is required before some clearer conclusions can be drawn.

There are some more general extensions to the system which may be of interest.

Incremental abstract interpretation Some analysis can be used during construction of programs. It could be interesting to obtain information about strictness and binding time when the program is to be used for lazy evaluation or partial evaluation. There is a fast growing literature about incremental evaluation of attribute grammars and some of these ideas may be applicable to abstract interpretation in this setting.

Theorem prover The correctness proof of the attribute grammar is performed independently of the system. It may, however, be possible to use some automatic or manual theorem prover to verify continuity of functions and that relations are inductive. This is likely to give a higher degree of reliability of analyses developed in the system.

There is an area in the connection between attribute grammars and abstract interpretation we have not considered.

Analysis of attribute grammars It may be possible to use abstract interpretation to analyse attribute grammars for a number of properties. Some possibilities are sharing of attribute values and evaluation order (strictness, liveness,..). Attribute grammars is essentially a lazy programming language and many techniques for analysis of functional programming may be used in the implementation of attribute grammars.

References

The bibliography lists papers referenced in the thesis and under each reference it is listed on which pages in the thesis it is cited. A few papers central to the subject, not referenced directly in the thesis, is also included in the bibliography.

Some conference titles and names of journals are abbreviated in the bibliography. The abbreviations are explained below.

C.ACM Communications of the ACM.

ESOP European Symposium on Programming.

FPCA Functional Programming and Computer Architecture.

ICALP Int. Coll. on Automata, Languages and Programming.

J.ACM Journal of the ACM.

LNCS Lecture Notes in Computer Science.

MFCS Mathematical Foundation of Computer Science.

PLILP Programming Language Implementation and Logic Programming.

POPL Principles of Programming Languages.

TOPLAS ACM Transactions Programming Languages and Systems.

WAGA Workshop on Attribute Grammars and Applications.

[Abramsky 1986] S Abramsky. *Strictness analysis and polymorphic invariance*. In *Programs as Data Objects* (H Ganzinger and N D Jones, eds.), pp. 1–23. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.
Referenced on page 6.

[Abramsky & Hankin 1987] S Abramsky and C Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis-Horwood, 1987.
Referenced on pages 8, 45.

[Abramsky 1990] S Abramsky. *Abstract interpretation, logical relations and Kan extensions*. Journal of logic and computation **1**(1), 1990.
Referenced on page 7.

[Aho, Sethi & Ullman 1986] A V Aho, R Sethi and J D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
Referenced on pages 9, 29, 90, 92, 94, 109.

[Allen & Cocke 1976] F E Allen and J A Cocke. *A program data flow analysis procedure*. C. ACM **19**(3), pp. 137–147, 1976.

[Apt & Delporte-Gallet 1986] K R Apt and C Delporte-Gallet. *Syntax-Directed Analysis of Liveness Properties of While Programs.* Inform. and Control **68**(1-3), pp. 223–253, Jan., 1986.

[Babich & Jazayeri 1978] W A Babich and M Jazayeri. *The Method of Attributes for Data Flow Analysis (part 1 and 2).* Acta Inf. **10**, pp. 245–272, 1978.
Referenced on pages 10, 92.

[Barendregt 1984] H P Barendregt. *The Lambda Calculus - Its Syntax and Semantics.* Volume 103 of Studies in logic and the foundation of mathematics, Revised edition. North-Holland, 1984.
Referenced on pages 63, 64.

[Blikle & Tarlecki 1983] A Blikle and A Tarlecki. *Naive denotational semantics.* In *IFIP'83, Paris, France*, pp. 345–355. North-Holland, Sept., 1983.

[Bloss & Hudak 1986] A Bloss and P Hudak. *Variations on Strictness Analysis.* In *LISP'86, Cambridge, Mass*, pp. 132–142, Aug., 1986.
Referenced on page 6.

[Bloss & Hudak 1987] A Bloss and P Hudak. *Path Semantics.* In *Mathematical Foundation of Programming Language Semantics'87, New Orleans, Louisiana* (M Mislove, ed.), pp. 476–489. Volume 298 of LNCS. Springer-Verlag, Apr., 1987.
Referenced on page 6.

[Bondorf et al 1988] A Bondorf, N D Jones, T Mogensen and P Sestoft. *Binding time analysis and the taming of self-application.* Tech. Rep. DIKU, Univ. of Copenhagen, Denmark, 1988.
Referenced on page 7.

[Bourdoncle 1990] F Bourdoncle. *Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity.* In *PLILP'90*, pp. 307–323. Volume 456 of LNCS. Springer-Verlag, 1990.
Referenced on page 7.

[Bruynooghe et al 1987] M Bruynooghe, B Demoen, A Callebaut and G Janssens. *Abstract interpretation : Towards the global optimization of Prolog programs.* In *IEEE Symposium on Logic Programming.* IEEE, 1987.
Referenced on page 7.

[Burn, Hankin & Abramsky 1986] G L Burn, C L Hankin and S Abramsky. *The Theory of Strictness Analysis for Higher Order Functions.* In *Programs as Data Objects* (H Ganzinger and N D Jones, eds.), pp. 42–62. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.
Referenced on pages 6, 45.

[Chirica & Martin 1979] L M Chirica and D F Martin. *An Order-Algebraic Definition of Knuthian Semantics.* Math. Systems Theory **13**, pp. 1–27, 1979.
Referenced on pages 8, 88, 89, 91, 95.

[Clack & Jones 1985] C Clack and S P Jones. *Strictness analysis—a practical approach*. In *FPCA'85, Nancy, France*, pp. 35–49. Volume 201 of LNCS. Springer-Verlag, Sept., 1985.
Referenced on pages 71, 74.

[Courcelle & Deransart 1988] B Courcelle and P Deransart. *Proofs of partial correctness for attribute grammars with applications to recursive procedures and logic programming*. Inform. and Comput. **78**(1), pp. 1–55, July, 1988.
Referenced on pages 8, 91.

[Cousot & Cousot 1977] P Cousot and R Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In *4th POPL, Los Angeles, CA*, pp. 238–252, Jan., 1977.
Referenced on pages 5, 7, 37, 46, 59, 63, 96.

[Cousot & Cousot 1979] P Cousot and R Cousot. *Systematic Design of Program Analysis Frameworks*. In *6th POPL, San Antonio, Texas*, pp. 269–282, Jan., 1979.
Referenced on page 5.

[Cousot 1981] P Cousot. *Semantics Foundation of Program Analysis*. In *Program Flow Analysis: Theory and Applications* (S S Muchnick and N D Jones, eds.), chapter 10, pp. 303–342. Prentice-Hall, 1981.
Referenced on page 5.

[Damas & Milner 1982] L Damas and R Milner. *Principal type-schemes for functional programs*. In *9th POPL, Albuquerque, NM*, pp. 207–212, Jan., 1982.
Referenced on page 6.

[Deransart 1983] P Deransart. *Logical Attribute Grammars*. In *IFIP'83, Paris, France*, pp. 463–469. North-Holland, Sept., 1983.
Referenced on pages 8, 91, 94, 96, 105.

[Deransart & Maluszynski 1985] P Deransart and J Maluszynski. *Relating Logic Programs and Attribute Grammars*. Tech. Rep. LITH-IDA-R-85-08. Linköping Univ., Apr., 1985.
Referenced on pages 8, 91.

[Deransart, Jourdan & Lorho 1988] P Deransart, M Jourdan and B Lorho. *Attribute grammars. Definitions, systems, and bibliography*. Volume 323 of LNCS. Springer-Verlag, 1988.
Referenced on pages 8, 9, 91, 92, 94, 96.

[Despeyroux 1986] J Despeyroux. *Proof of Translation in Natural Semantics*. Research Report 514. Rocquencourt, Apr., 1986.
Referenced on page 7.

[Deutsch 1990] A Deutsch. *On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications.* In *17th POPL, San Fransisco, California*, pp. 157–168. ACM Press, Jan., 1990.
Referenced on page 7.

[Engelfriet 1984] J Engelfriet. *Attribute Grammars: Attribute Evaluation Methods.* In *Methods and Tools for Compiler Construction* (B Lorho, ed.), pp. 103–138. Cambridge Univ. Press, 1984.
Referenced on pages 9, 94.

[Farrow 1986] R Farrow. *Automatic Generation of Fixed-Point-Finding Evaluators for Circular but Well-Defined Attribute Grammars.* In *SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, California*, pp. 85–98. Volume 21(7) of ACM SIGPLAN Not., July, 1986.
Referenced on page 9.

[Filé 1986] G Filé. *Machines for Attribute Grammars.* Inform. and Control **69**, pp. 41–124, 1986.
Referenced on page 9.

[Foderaro, Sklower & Layer 1983] J K Foderaro, K L Sklower and K Layer. *The Franz Lisp manual.* Univ. of California, 1983.
Referenced on page 73.

[Fosdick & Osterweil 1976] L D Fosdick and L J Osterweil. *Data Flow Analysis in Software Reliability.* Comp. Surv. **8**(4), pp. 305–330, Sept., 1976.
Referenced on page 5.

[Gallier 1984] J Gallier. *An Efficient Evaluator for Attribute Grammars with Conditionals.* Tech. Rep. MS-CIS-83-36. Philadelphia, PA, May, 1984.
Referenced on pages 9, 91, 110.

[Gallier, Manion & McEnerney 1985] J H Gallier, F J Manion and J McEnerney. *A Compiler Generator based on Attribute Evaluation.* Tech. Rep. MS-CIS-85-59. Philadelphia, PA, Oct., 1985.
Referenced on pages 9, 92, 94.

[Ganzinger 1980] H Ganzinger. *Transforming denotational semantics into practical attribute grammars.* In *Semantics-Directed Compiler Generation* (N D Jones, ed.), pp. 1–69. Volume 94 of LNCS. Springer-Verlag, Jan., 1980.
Referenced on page 8.

[Ganzinger & Giegerich 1984] H Ganzinger and R Giegerich. *Attribute Coupled Grammars.* In *SIGPLAN '84 Symposium on Compiler Construction, Montreal, Canada*, pp. 157–170. Volume 19(6) of ACM SIGPLAN Not., June, 1984.
Referenced on pages 9, 93.

[Giegerich 1988] R Giegerich. *Composition and Evaluation of Attribute Coupled Grammars.* Acta Inf. **25**, pp. 355–423. Springer-Verlag, 1988.
Referenced on page 9.

[Goguen et al 1977] J A Goguen, J W Thatcher, E G Wagner and J B Wright. *Initial algebra semantics and continuous algebras*. J. ACM. **24**(1), 1977.
Referenced on page 91.

[Gordon 1979] M J C Gordon. *The Denotational Description of Programming Languages: An Introduction*. Springer-Verlag, 1979.
Referenced on pages 8, 35, 68, 98.

[Gunter, Mosses & Scott 1989] C A Gunter, P D Mosses and D S Scott. *Semantic Domains and Denotational Semantics*. Tech. Rep. PB-276. DAIMI, Univ. of Aarhus, Denmark, Apr., 1989.
Referenced on pages 15, 32.

[Hecht 1977] M Hecht. *Flow analysis of computer programs*. North-Holland, 1977.
Referenced on pages 5, 38.

[Hudak & Bloss 1985] P Hudak and A Bloss. *The aggregate update problem in functional programming systems*. In *12th POPL, New Oleans, Lousiana*, pp. 300–314, Jan., 1985.
Referenced on page 6.

[Hudak 1987] P Hudak. *A semantic model of reference counting and its abstraction*. In *Abstract Interpretation of Declarative Languages* (S Abramsky and C Hankin, eds.), chapter 3, pp. 45–62. Ellis-Horwood, 1987.
Referenced on page 6.

[Hudak & Young 1988] P Hudak and J Young. *A Collecting Interpretation of Expressions (Without Powerdomains) -Preliminary Report-*. In *15th POPL, San Diego, California*, pp. 107–118. ACM Press, Jan., 1988.
Referenced on page 6.

[Hughes 1985] J Hughes. *Lazy memo-functions*. In *FPCA'85, Nancy, France*. Volume 201 of LNCS. Springer-Verlag, Sept., 1985.
Referenced on page 73.

[Hughes 1988] J Hughes. *Backward Analysis of Functional Programs*. In *Proceedings of the workshop on Partial Evaluation and Mixed Computation* (D Bjørner et al., eds.), pp. 187–208. North-Holland, 1988.
Referenced on page 6.

[Hunt 1989] S Hunt. *Frontiers and open sets in abstract interpretation*. In *FPCA'89, London, England*, pp. 1–13. ACM Press, Sept., 1989.
Referenced on pages 7, 71.

[Irons 1961] E T Irons. *A syntax directed compiler for ALGOL60*. C. ACM **4**, pp. 51–55, 1961.
Referenced on pages 8, 85.

[Jayaraman & Plaisted 1987] Jayaraman and Plaisted. *Functional programming with sets*. In *FPCA'87, Portland, Oregon* (G Kahn, ed.), pp. 194–211. Volume 274 of LNCS. Springer-Verlag, Sept., 1987.
Referenced on page 79.

[Jensen & Mogensen 1990] T Jensen and T Mogensen. *A backwards analysis for compile time garbage collection*. In *ESOP'90, Copenhagen, Denmark*, pp. 227–239. Volume 432 of LNCS. Springer-Verlag, 1990.
Referenced on page 6.

[Johnson 1975] S C Johnson. *YACC: Yet Another Compiler-Compiler*. Manual. Murray Hill, NJ, 1975.
Referenced on pages 10, 90.

[Johnsson 1987] T Johnsson. *Attribute Grammars and Functional Programming*. In *FPCA'87, Portland, Oregon* (G Kahn, ed.), pp. 154–173. Volume 274 of LNCS. Springer-Verlag, Sept., 1987.
Referenced on page 9.

[Jones & Muchnick 1981] N D Jones and S S Muchnick. *Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra*. In *Program Flow Analysis: Theory and Applications* (S S Muchnick and N D Jones, eds.), chapter 12, pp. 380–393. Prentice-Hall, 1981.
Referenced on pages 38, 40.

[Jones, Sestoft & Søndergaard 1985] N D Jones, P Sestoft and H Søndergaard. *An Experiment in Partial Evaluation: The Generation of a Compiler Generator*. In *Rewriting Techniques and Applications*, pp. 124–140. Volume 202 of LNCS. Springer-Verlag, 1985.
Referenced on pages 7, 93.

[Jones & Mycroft 1986] N D Jones and A Mycroft. *Data Flow Analysis of Applicative Programs using Minimal Function Graphs*. In *13th POPL, St. Petersburg, Florida*, pp. 296–306, Jan., 1986.
Referenced on pages 35, 41, 42, 71, 73, 105.

[Jones & Simon 1986] L G Jones and J Simon. *Hierarchical VLSI Design Systems Based on Attribute Grammars*. In *13th POPL, St. Petersburg, Florida*, pp. 58–69, Jan., 1986.
Referenced on page 9.

[Jones & Clack 1987] S P Jones and C Clack. *Finding fixpoints in abstract interpretation*. In *Abstract Interpretation of Declarative Languages* (S Abramsky and C Hankin, eds.), chapter 11, pp. 246–265. Ellis-Horwood, 1987.
Referenced on pages 7, 71.

[Jones 1987a] N D Jones. *Flow Analysis of Lazy Higher Order Functional Programs*. In *Abstract Interpretation of Declarative Languages* (S Abramsky and C Hankin, eds.), chapter 5, pp. 103–122. Ellis-Horwood, 1987.
Referenced on page 7.

[Jones 1987b] S T P Jones. *The implementation of functional programming languages*. Prentice-Hall, 1987.
Referenced on pages 7, 71, 74.

[Jones, Sestoft & Søndergaard 1989] N D Jones, P Sestoft and H Søndergaard. *Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation.* Lisp and Symbolic Computation **2**(1), pp. 9–50, 1989.
Referenced on page 7.

[Jourdan 1984] M Jourdan. *Recursive Evaluators for Attribute Grammars: An Implementation.* In *Methods and Tools for Compiler Construction* (B Lorho, ed.), pp. 139–163. Cambridge Univ. Press, 1984.
Referenced on page 9.

[Jourdan, Bellec & Parigot 1990] M Jourdan, C L Bellec and D Parigot. *The OLGA Attribute Grammar Description Language.* In *WAGA'90* (P Deransart and M Jourdan, eds.), pp. 222–237. Volume 461 of LNCS. Springer-Verlag, Oct., 1990.
Referenced on page 10.

[Jouvelot 1986] P Jouvelot. *Designing new languages and new language manipulation systems using ML.* ACM SIGPLAN Not. **21**(8), pp. 40–52, Aug., 1986.
Referenced on page 10.

[Jouvelot 1987] P Jouvelot. *Semantic Parallelization: a practical execise in abstract interpretation.* In *14th POPL, Munich, West Germany*, pp. 39–48, Jan., 1987.

[Kahn 1987] G Kahn. *Natural Semantics.* Research Report 601. Rocquencourt, Feb., 1987.

[Kastens 1984] U Kastens. *The GAG-System—A Tool for Compiler Construction.* In *Methods and Tools for Compiler Construction* (B Lorho, ed.), pp. 165–182. Cambridge Univ. Press, 1984.
Referenced on page 10.

[Kieburtz & Napierala 1987] R B Kieburtz and M Napierala. *Abstract Semantics.* In *Abstract Interpretation of Declarative Languages* (S Abramsky and C Hankin, eds.), chapter 7, pp. 143–180. Ellis-Horwood, 1987.

[Kildall 1973] G Kildall. *A Unified Approach to Global Program Optimization.* In *1st POPL*, pp. 194–206, Oct., 1973.
Referenced on page 5.

[Knuth 1968] D E Knuth. *Semantics of Context-Free Languages.* Math. Systems Theory **2**(2), pp. 127–145, June, 1968. Correction ibid 5(1):95–96 Mar. 1971.
Referenced on pages 8, 85, 86, 89, 91, 110.

[Knuth 1971] D E Knuth. *Examples of formal semantics.* In *Symposium on semantics of algorithmic languages* (E Engeler, ed.), pp. 212–235. Volume 188 of Lect. Notes in Math. Springer-Verlag, 1971.
Referenced on pages 8, 92.

[Lee et al 1988] P Lee et al. *The Ergo Support System: An Integrated Set of Tools for Prototyping Integrated Environments.* In *SIGPLAN '88 Conference on PLDI, Atlanta, Georgia*, pp. 25–34. Volume 23(7) of ACM SIGPLAN Not., July, 1988.
Referenced on page 10.

[Madsen 1980] O L Madsen. *On Defining Semantics by means of Extended Attribute Grammars.* In *Semantics-Directed Compiler Generation* (N D Jones, ed.), pp. 259–299. Volume 94 of LNCS. Springer-Verlag, Jan., 1980.
Referenced on pages 9, 91.

[Manna, Ness & Vuillemin 1972] Z Manna, S Ness and J Vuillemin. *Inductive methods for proving properties of programs.* In *ACM Conference on Proving Assertions About Programs*, pp. 27–50. Volume 7(1) of ACM SIGPLAN Not., Jan., 1972.
Referenced on pages 7, 23, 24, 103.

[Marlowe & Ryder 1990] T J Marlowe and B G Ryder. *Properties of data flow frameworks - a unified model.* Acta Inf. **28**(2), pp. 121–163, Dec., 1990.
Referenced on page 5.

[Marriott & Søndergaard 1989] K Marriott and H Søndergaard. *Semantics-based dataflow analysis of logic programs.* In *IFIP'89*. North-Holland, 1989.
Referenced on page 7.

[Martin & Hankin 1987] C Martin and C Hankin. *Finding Fixed Points in Finite Lattices.* In *FPCA'87, Portland, Oregon* (G Kahn, ed.), pp. 426–445. Volume 274 of LNCS. Springer-Verlag, Sept., 1987.
Referenced on pages 7, 71.

[Mayoh 1981] B H Mayoh. *Attribute Grammars and Mathematical Semantics.* SIAM J. Comp. **10**(3), pp. 503–518, Aug., 1981.
Referenced on pages 8, 91.

[Michaelson 1986] G Michaelson. *Interpreters from Functions and Grammars.* Computer Languages **11**(2), pp. 85–104, 1986.
Referenced on page 93.

[Morel 1984] E Morel. *Data Flow Analysis and Global Optimization.* In *Methods and Tools for Compiler Construction* (B Lorho, ed.), pp. 289–315. Cambridge Univ. Press, 1984.

[Mosses 1979] P Mosses. *SIS - Semantics Implementation System. Reference Manual and User Guide.* Tech. Rep. MD-30. DAIMI, Univ. of Aarhus, Denmark, Aug., 1979.
Referenced on page 68.

[Mosses 1983] P Mosses. *Abstract Semantic Algebras!* In *Formal Description of Programming Concepts* (D Bjørner, ed.), pp. 63–88. North-Holland, 1983.
Referenced on page 93.

[Muchnick & Jones 1981] S S Muchnick and N D Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
Referenced on page 5.

[Mulmuley 1985] K Mulmuley. *Full abstraction and semantics equivalence*. Ph.D. Thesis CMU-CS-85-148. Carnegie-Mellon Univ., 1985.
Referenced on page 7.

[Mycroft 1980] A Mycroft. *The Theory and Practice of Transforming Call-by-Need into Call-by-Value*. In *International Symposium on Programming'80, Paris, France* (B Robinet, ed.), pp. 269–281. Volume 83 of LNCS. Springer-Verlag, Apr., 1980.
Referenced on pages 6, 52.

[Mycroft 1981] A Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. Ph.D. Thesis. Univ. of Edinburgh, Dec., 1981.
Referenced on pages 5, 37, 38, 45, 48.

[Mycroft & Nielson 1983] A Mycroft and F Nielson. *Strong Abstract Interpretation using Power Domains*. In *ICALP'83, Barcelona, Spain*, pp. 536–547. Volume 154 of LNCS. Springer-Verlag, July, 1983.

[Mycroft & Jones 1986] A Mycroft and N D Jones. *A Relational Framework for Abstract Interpretation*. In *Programs as Data Objects* (H Ganzinger and N D Jones, eds.), pp. 156–171. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.
Referenced on pages 7, 25.

[Mycroft 1987] A Mycroft. *A Study on Abstract Interpretation and "Validating Microcode Algebraically"*. In *Abstract Interpretation of Declarative Languages* (S Abramsky and C Hankin, eds.), chapter 9, pp. 199–218. Ellis-Horwood, 1987.

[Naur 1965] P Naur. *Checking of Operand Types in Algol Compilers*. BIT **5**, pp. 151–163, 1965.
Referenced on page 5.

[Nielson 1984] F Nielson. *Abstract Interpretation using Domain Theory*. Ph.D. Thesis. Univ. of Edinburgh, Oct., 1984.
Referenced on pages 5, 6, 7.

[Nielson & Nielson 1986] H R Nielson and F Nielson. *Semantics Directed Compiling for Functional Languages*. In *LISP'86, Cambridge, Mass*, pp. 249–257, Aug., 1986.
Referenced on page 91.

[Nielson 1986a] F Nielson. *Abstract Interpretation of Denotational Definitions*. In *STACS'86*, pp. 1–20. Volume 210 of LNCS. Springer-Verlag, 1986.
Referenced on page 6.

[Nielson 1986b] F Nielson. *Expected Forms of Data Flow Analysis*. In *Programs as Data Objects* (H Ganzinger and N D Jones, eds.), pp. 172–191. Volume 217 of LNCS. Springer-Verlag, Oct., 1986.

[Nielson 1987] H R Nielson. *The core of the PSI-system (version 1.0)*. Tech. Rep. IR-87-02. Aalbog Universitetscenter, Mar., 1987.
Referenced on pages 7, 68, 109.

[Nielson 1988] F Nielson. *Strictness Analysis and Denotational Abstract Interpretation*. Inform. and Comput. **76**, pp. 29–92, 1988.
Referenced on page 78.

[Nielson & Nielson 1988] F Nielson and H R Nielson. *Two-level semantics and code generation*. Theor. Comp. Sci. **56**(1), pp. 59–133, Jan., 1988.
Referenced on pages 7, 72, 93, 110, 111.

[Nielson 1989] F Nielson. *Two-level semantics and abstract interpretation*. Theor. Comp. Sci. **69**, pp. 117–242, 1989.
Referenced on pages 6, 78, 109.

[Nord & Pfenning 1989] R L Nord and F Pfenning. *The Ergo Attribute System*. SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments **24**(2), pp. 110–120, Feb., 1989.
Referenced on page 10.

[O'Keefe 1987] R A O'Keefe. *Finite Fixed-Point Problems*. In *International Conference on Logic Programming*, pp. 729–743, May, 1987.

[Paulson 1982] L Paulson. *A Semantics-Directed Compiler Generator*. In *9th POPL, Albuquerque, NM*, pp. 224–233, Jan., 1982.
Referenced on pages 9, 10, 92, 93, 110.

[Paulson 1984] L Paulson. *Compiler Generation from Denotational Semantics*. In *Methods and Tools for Compiler Construction* (B Lorho, ed.), pp. 219–250. Cambridge Univ. Press, 1984.
Referenced on page 10.

[Pfenning & Elliott 1988] F Pfenning and C Elliott. *Higher-Order Abstract Syntax*. In *SIGPLAN '88 Conference on PLDI, Atlanta, Georgia*, pp. 199–208. Volume 23(7) of ACM SIGPLAN Not., July, 1988.
Referenced on page 9.

[Phoa 1990] W Phoa. *Effective Domains and Intrinsic Structure*. In *LICS'90*. IEEE, June, 1990.
Referenced on page 64.

[Plotkin 1981] G D Plotkin. *A Structural Approach to Operational Semantics*. Tech. Rep. FN-19. DAIMI, Univ. of Aarhus, Denmark, Sept., 1981.

[Plotkin 1982] G Plotkin. *Domain Theory*. Univ. of Edinburgh, 1982.
Referenced on pages 8, 13, 23.

[Reps & Teitelbaum 1984] T Reps and T Teitelbaum. *The Synthesizer Generator*. In *SIGPLAN '84 Symposium on Compiler Construction, Montreal, Canada*, pp. 42–48. Volume 19(6) of ACM SIGPLAN Not., June, 1984.
Referenced on pages 10, 92, 93, 94.

[Reynolds 1974] J C Reynolds. *On the relation between direct and continuation semantics*. In *ICALP'74*, pp. 141–156. Volume 14 of LNCS. Springer-Verlag, 1974.
Referenced on page 7.

[Reynolds 1983] J C Reynolds. *Types, abstraction and parametric polymorphism*. In *IFIP'83, Paris, France*. North-Holland, Sept., 1983.
Referenced on pages 7, 8, 23, 53, 58, 105.

[Rosen 1980] B K Rosen. *Monoids for rapid data flow analysis*. SIAM J. Comp. **9**, pp. 159–196, 1980.
Referenced on page 5.

[Rosendahl 1989] M Rosendahl. *Automatic Complexity Analysis*. In *FPCA'89, London, England*, pp. 144–156. ACM Press, Sept., 1989.
Referenced on pages 6, 15, 43, 59, 83.

[Rosendahl 1990] M Rosendahl. *Abstract Interpretation using Attribute Grammars*. In *WAGA'90* (P Deransart and M Jourdan, eds.), pp. 143–156. Volume 461 of LNCS. Springer-Verlag, Oct., 1990.
Referenced on page 95.

[Sagiv et al 1989] S Sagiv, O Edelstein, N Francez and M Rodeh. *Resolving circularity in attribute grammars with application to data flow analysis*. In *16th POPL, Austin, Texas*, pp. 36–46. ACM Press, Jan., 1989.
Referenced on page 9.

[Schmidt 1982] D A Schmidt. *Denotational semantics as a programming language*. Internal Report CSR-100-82. Univ. of Edinburgh, Jan., 1982.
Referenced on page 68.

[Schmidt 1986] D A Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Newton, MA, 1986.
Referenced on pages 8, 13, 15, 32, 35, 68.

[Smyth & Plotkin 1982] M B Smyth and G Plotkin. *The category-theoretic solution of recursive domain equations*. SIAM J. Comp. **11**(4), pp. 761–783, Nov., 1982.
Referenced on pages 8, 23.

[Statman 1980] R Statman. *Completeness, invariance, and λ-definability*. J. Symbolic Logic **47**(1), pp. 17–26, 1980.
Referenced on page 7.

[Statman 1985] R Statman. *Logical relations and the Typed λ-Calculus*. Inform. and Control **65**(2/3), pp. 85–97, May, 1985.
Referenced on page 7.

[Stoy 1977] J E Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
Referenced on pages 7, 8, 35, 62, 68, 103.

[Tarski 1955] A Tarski. *A lattice-theoretical fixpoint theorem and its applications*. Pacific J. Math. **5**, pp. 285–309, 1955.
Referenced on page 23.

[Uddeborg 1988] G O Uddeborg. *A Functional Parser Generator*. Tech. Rep. 43. Dept. of Comp. Sc., Univ. of Göteborg, Chalmers Univ. of Tech., Feb., 1988.
Referenced on page 10.

[Vogt, Swierstra & Kuiper 1989] H H Vogt, S D Swierstra and M F Kuiper. *Higher order attribute grammars*. In *SIGPLAN '89 Conference on PLDI, Portland, Oregon*, pp. 131–145. Volume 24(7) of ACM SIGPLAN Not., July, 1989.
Referenced on page 9.

[Wadler 1987] P Wadler. *Strictness analysis on non-flat domains (by abstract interpretation)*. In *Abstract Interpretation of Declarative Languages* (S Abramsky and C Hankin, eds.), chapter 12, pp. 266–275. Ellis-Horwood, 1987.
Referenced on page 6.

[Wadler & Hughes 1987] P Wadler and R J M Hughes. *Projections for Strictness Analysis*. In *FPCA'87, Portland, Oregon* (G Kahn, ed.), pp. 385–407. Volume 274 of LNCS. Springer-Verlag, Sept., 1987.
Referenced on pages 6, 7.

[Wadsworth 1978] C Wadsworth. *Lecture Notes in Domain Theory*. Lect. Notes. Univ. of Edinburgh, 1978.
Referenced on pages 8, 13.

[Watt & Madsen 1983] D A Watt and O L Madsen. *Extended Attribute Grammars*. Comput. J. **26**(2), pp. 142–153, 1983.
Referenced on pages 9, 86.

[Wegbreit 1975] B Wegbreit. *Property extraction in well-founded property sets*. IEEE Trans. Software Engrg. **1**, pp. 270–285, 1975.
Referenced on page 5.

[Wilhelm 1981] R Wilhelm. *Global Flow Analysis and Optimization in the MUG2 Compiler Generating System*. In *Program Flow Analysis: Theory and Applications* (S S Muchnick and N D Jones, eds.), chapter 5, pp. 132–159. Prentice-Hall, 1981.
Referenced on pages 10, 92.

[Young & Hudak 1986] J Young and P Hudak. *Finding fixpoints on functional spaces*. Tech. Rep. YALEEU/DCS/RR-505. Yale Univ., Dec., 1986.
Referenced on pages 7, 71.

Index

abstract interpretation, 3, 5
abstract semantics, 3
abstract syntax, 32
abstract syntax definition, 32
abstraction, 7, 46
admissible, 24
algebraic cpo, 15
analysed, 3
applied positions, 87
assignment statement, 85
attribute definitions, 2
Attribute evaluation, 85
attribute grammar, 2
attributed productions, 87
attributed scheme, 33, 92, 94, 96
attributes, 85

binding time analysis, 7
boolean values, 15

Cartesian product, 16
Category, 3
chain, 14
chain-complete partial order, 14
circular attribute grammar, 85
coalesced sum, 18
collecting semantics, 3, 37
complete, 24
complete lattice, 23
complete partially ordered set, 3, 14
concrete syntax definition, 32
concretisation, 7, 46
consistent set of attribute values, 85
constant propagation, 49
context-free grammar, 30
continuation semantics, 3
continuous, 16
copy rules, 86
cpo, 3, 14

data flow analysis, 5
data flow information, 3
defining positions, 87
Domain, 3
domain, 15
domain assignment, 55
Domain attribute grammars, 95
domain attribute grammars, 95
domain expression, 19

effective domains, 64
embedded, 20
embedding, 20

finite height, 15
finite lists, 19
fixpoint, 16
fixpoint induction, 35, 48
fixpoint semantics, 35
flat, 15
formalisms, 3
frontier, 71
function domain, 18

Galois connection, 46

identifiers, 15
inclusive relations, 23
independent attribute method, 38
induced analysis, 48
inductive, 24
infinite lists, 19
infinite sum, 18

inherited attribute, 85
input-output function, 37
instrumented semantics, 43
interpretation, 57, 96
inverse limit, 21
isotone, 16

join operation, 38

language, 29
lazy fixpoint iteration, 11
least element, 14
least fixpoint, 16
least number operator, 64
least upper bound, 14
left hand side, 30
lifted, 18
limit, 14
limit-preserving maps, 16
literals, 15
logical relations, 24
lub, 14

meaning assignment, 56
meet operation, 38
meet over all path, 38
memo-function, 72
minimal function graph, 41
monotonic, 16

natural numbers, 15
nonterminal, 29, 30

order-preserving, 16

parse, 32
parse trees, 31
parser, 33
partial order, 14
partially known structure, 58
pending analysis, 71
pointed cpo, 14
poset, 14
power domains, 45
power set, 18
productions, 30
program analysis, 3
program analysis problems, 3
program point, 38
program transformation, 3
projection, 20
projections, 7

recognise, 29
relation, 13
relation assignment, 57
relational method, 40
retraction pairs, 46
retraction-pair, 20
right hand side, 30
root symbol, 30

semantic domain, 15
semantic grammars, 110
separated sum, 17, 18
solution techniques, 3
specification, 105
Standard semantics, 3
standard semantics, 3
static semantics, 37
strict, 16
strictness analysis, 6
syntactic category, 3
synthesised attribute, 85

terminal, 29
terminals, 30
termination, 63
the ascending Kleene sequence, 16
tree grammar, 32
two-level metalanguage, 109
type assignment, 54

unambiguous, 29
upper bound, 14

valuation, 105
version, 57, 105

well-formed attribute grammar, 86

widening, 63