Basic Research in Computer Science

Trust and Dependence Analysis

Peter Ørbæk

BRICS Dissertation Series **DS-97-2**

ISSN 1396-7002 **July 1997**

Copyright © 1997, BRICS, Department of Computer Science University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/97/2/`

Trust and Dependence Analysis

Peter Ørbæk

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

Trust and Dependence Analysis

Dissertation
presented to the Faculty of Science
of the University of Aarhus
in partial fulfillment of the requirements for the
Ph.D. degree

by
Peter Ørbæk
last revised July 17, 1997

Abstract

The two pillars of *trust analysis* and *dependence algebra* form the foundation of this thesis. Trust analysis is a static analysis of the run-time trustworthiness of data. Dependence algebra is a rich abstract model of data dependences in programming languages, applicable to several kinds of analyses.

We have developed trust analyses for imperative languages with pointers as well as for higher order functional languages, utilizing both abstract interpretation, constraint generation and type inference.

The mathematical theory of dependence algebra has been developed and investigated. Two applications of dependence algebra have been given: a practical implementation of a trust analysis for the C programming language, and a soft type inference system for action semantic specifications. Soundness results for the two analyses have been established.

Acknowledgments

Many people have contributed in various ways to this thesis. First I would like to thank my advisor, Peter D. Mosses, for keeping his faith in me throughout my studies and allowing me to pursue different avenues of research. I must also thank Jens Palsberg for introducing me to action semantics many years ago, for working with me on Chapter 3 of the present thesis, and for many discussions and good advise. I must also thank Albert R. Meyer for hosting my visit to the LCS at MIT. Many people have contributed good advise, commented on drafts and much more. I hereby express my gratitude to all of them, and in particular to: Gerth S. Brodal, Allan Cheng, Olivier Danvy, Dirk Dussart, and Søren B. Lassen. I must also express my gratitude to my thesis committee: especially David Schmidt and Torben Mogensen, for their detailed comments that helped improve this final version. Finally, I must thank the BRICS project for funding and for providing an excellent scientific environment.

Contents

1 Introduction **1**
1.1 Trust Analysis 1
1.1.1 Why Trust Analysis isn't... 5
1.2 Dependence Analysis 5
1.3 Overview of the Thesis 7
1.3.1 Simple Trust Analysis 7
1.3.2 Higher-Order Trust Analysis 8
1.3.3 Dependence Algebra 8
1.3.4 Trust Analysis of C Programs 9
1.3.5 Soft Type Inference for Action Semantics 9
1.4 Acknowledgments 10

2 Can you Trust your Data? **11**
2.1 Introduction 11
2.2 Motivation 11
2.2.1 The Sendmail example 12
2.3 The `While` language 13
2.4 Standard Semantics 13
2.5 Instrumented Semantics 14
2.6 Abstract Interpretation 16
2.7 Constraint Generation 18
2.8 Extensions 22
2.9 Conclusion 22
2.10 Safety of Abstract Interpretation 23
2.11 Safety of Constraint Generation 25

3 Trust in the λ-calculus **29**
3.1 Introduction 29
3.1.1 Intuitions and Motivation 29
3.1.2 An Example 30
3.2 Syntax and Semantics 31
3.2.1 Reduction Rules 32
3.2.2 The Nature of check 32
3.2.3 Church-Rosser 35
3.2.4 Denotational Semantics 40
3.3 The Type System 42

3.3.1 Rules . . . 43
3.3.2 Subject Reduction . . . 44
3.3.3 Comparison with the Curry System . . . 45
3.3.4 Simulation . . . 47
3.3.5 Strong Normalization . . . 51
3.4 Type Inference . . . 51
3.4.1 Constraints . . . 51
3.4.2 Algorithm . . . 54
3.5 Extensions . . . 56
3.6 Related Work . . . 58
3.7 Summary . . . 58
3.7.1 Acknowledgements . . . 58

4 Dependence Algebra 59
4.1 Introduction . . . 59
4.1.1 Slots . . . 59
4.1.2 The Meaning of Dependence . . . 60
4.2 Basic Definitions . . . 62
4.2.1 Limits . . . 63
4.2.2 Vectors . . . 64
4.2.3 Matrices . . . 66
4.2.4 Quadruples . . . 67
4.3 Image . . . 70
4.4 Pre-image . . . 71
4.5 Concatenation . . . 72
4.6 Quad Languages . . . 74
4.7 Approximating a Single Quad . . . 75
4.8 Abstracting Sets of Quads . . . 81
4.9 Soundness . . . 86
4.10 Constructing a **DAlg**-structure . . . 87
4.11 Reverse Images . . . 91
4.12 Discussion . . . 97
4.13 Summary . . . 99

5 Trust Analysis for C 101
5.1 Introduction . . . 101
5.2 Overview of the Implementation . . . 102
5.3 Parsing and Type Checking . . . 105
5.3.1 Syntax Extensions . . . 106
5.4 The Internal Representation . . . 107
5.5 The Stack Machine . . . 108
5.6 Semantics of the Flow Graph . . . 109
5.7 Compiling to Internal Representation . . . 110
5.7.1 Peephole Optimization . . . 113
5.7.2 Undetermined Evaluation Order . . . 113
5.8 Control Dependence and Decoration . . . 115
5.8.1 Decoration . . . 117

5.8.2 Computing Control Dependences . . . 119
5.9 The Static Call Graph . . . 120
5.10 The Trust Analysis Proper . . . 120
5.10.1 The Analysis . . . 121
5.10.2 Basic Soundness . . . 123
5.10.3 Sequences . . . 126
5.10.4 Calls and Returns . . . 128
5.11 Practical Efficiency . . . 130
5.12 Discussion . . . 131
5.13 Summary . . . 132

6 Soft Type Inference **133**
6.1 Motivation . . . 133
6.1.1 Related Work . . . 135
6.2 Introduction to Action Semantics . . . 136
6.2.1 Abstract Syntax . . . 137
6.2.2 Semantic Functions . . . 137
6.2.3 Semantic Entities . . . 138
6.3 A Subset of Action Notation . . . 138
6.4 A Natural Semantics . . . 139
6.4.1 Evaluation of Actions . . . 140
6.4.2 Evaluation of Yielders . . . 141
6.5 Simplification . . . 142
6.5.1 Simplification Algorithm . . . 144
6.6 The Dependence Algebra . . . 145
6.7 Translation . . . 147
6.8 Examples . . . 148
6.9 The Flow Analysis . . . 150
6.10 Soundness . . . 152
6.11 Finiteness . . . 159
6.12 Summary . . . 160

7 Conclusion **163**
7.1 Contributions . . . 163
7.2 Suggestions for Future Research . . . 164

Bibliography **165**

Index **172**

Chapter 1

Introduction

Static program analysis serves many purposes. Data flow analyses in optimizing compilers enable code optimizations, type inference algorithms help programmers find bugs at compile time, and security flow analyses enable certification of secure software. The program analyses presented in this thesis can be grouped under two main themes: *trust analysis*, and *dependence analysis*.

Trust analysis tracks the data flow in a program with the aim of ensuring that appropriate validation checks are made on all data paths leading to "dangerous" operations, such as entering data into a private database, deleting files, etc.

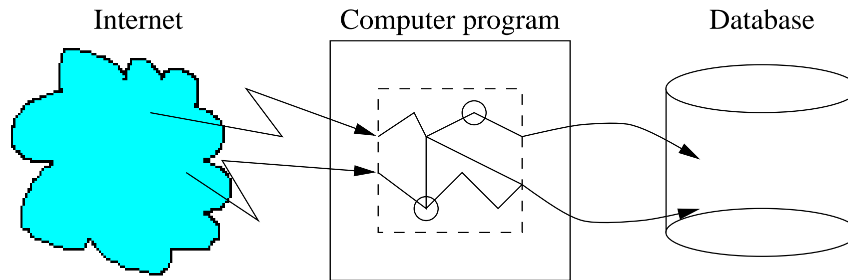
Dependence analysis is a more general framework that can be applied to different concrete analysis tasks. The data flow in a program is modeled as channels with certain properties determining which kinds of data can pass through the channels[1].

The following parts of the introduction give a more elaborate introduction to the two kinds of analysis and frame them in a historical context by discussing important previous work in the area.

1.1 Trust Analysis

Computers are increasingly being used to handle important transactions across the Internet. Legal documents and money orders are sent across the same network as is used to transfer e-mail and non-business related information. Thus separating these two kinds of information is becoming more and more of an issue. This problem is usually attacked using encryption and digital signatures for the important information flows between computers. But what if there, somewhere inside a program, is a data path between the reading of a message from the network and a dangerous operation depending on that message, such that the signature of the message is not checked on that path?

[1] The notion of dependence analysis as used in this thesis should not be confused with the notion of dependence of attribute sets in relational database theory.



An example of an application where it is important to perform validity checks is in an HTTP (Hypertext Transfer Protocol [BLFF96, FGM+97]) server. Most such servers allow separate programs (so-called CGI scripts, for Common Gateway Interface) to be started on the server machine in response to requests from clients. The usual convention is that the web browser requests an URL of the form "`http://www.company.com/cgi-bin/`*program*" to start *program*. A CGI script can in principle be any program, so it is important that the server process checks that the requested program is one of the few programs allowed to be executed in this manner. In a complicated server program there are many data paths from the reception of an URL to the command to be executed. It is important that the checks for allowed programs are done on *all* these data paths. Trust analysis has been developed with the goal that a compiler will be able to provide the programmer with a guarantee that checks are present on all data paths between the reception of untrusted input and the execution of commands where only trustworthy (checked, validated) information should be present.

Another example of where a trust analysis would be useful is in a part of the Gopher (an earlier and simpler distributed information service than the world-wide web [AML+93]) server, where through one of the provided gateways to other services one could contrive a special request to the server and thereby get arbitrary commands executed on the server machine, including starting a remote terminal window on that machine and thus thwarting all security measures. This security bug was later fixed by the developers when they were made aware of the problem. We believe that the use of a trust-analysis could have helped prevent this problem in the first place.

A third example of where trust analysis could be useful is in a web browser that must forbid the retrieval of certain URLs, for example to prevent children from viewing on-line pornography. Since there are many ways in a typical browser program to enter an URL, (the command line, configuration files, dialog boxes, ...) it's important that the checks for forbidden URLs are made on all the paths from reading user input to getting the URL from a server.

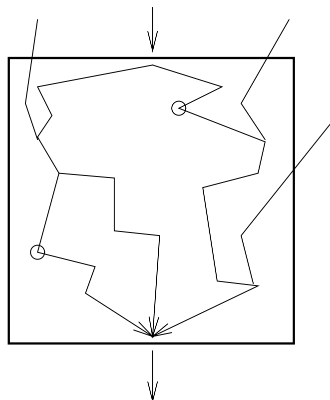
Since we want our analysis to be generally applicable, we do not consider the very specific tests that have to be done on input data in various situations, such as checking digital signatures or verifying pathnames against a known pattern. Devising these tests is still up to the careful program designer. Instead our analysis offers a "`trust`" construct that is meant to be applied to data after they have passed the specific validity checks, the analysis will then propagate this knowledge around the program. At points of the program where something dangerous is about to happen, such as starting another program or starting a transaction against a database, the programmer can write a "`check`" construct to ensure that the arguments can indeed be trusted, which means that they only depend on data that have actually passed the validity checks.

As an example of a run-time version of trust-checking, the programming language Perl [WS91] has a switch that turns on so-called *taint*-checks at run-time that will abort the program with an error message if tainted input data are being used in "dangerous" functions such as `unlink`. There's also as construct to convert tainted data to un-tainted form so that dangerous functions can be used on it. Our trust analysis is inspired by this feature of Perl, but our analysis is entirely static, so we can avoid run-time errors and overhead due to taint-checks.

A static check inevitably loses some flexibility as compared to run-time checking, but in the case of trust checks, it's not very useful to get a run-time trust violation, as such an error would typically occur too late in the data path for corrective action to be possible. So not much useful flexibility is lost. The goal of trust analysis is to make all promotions from untrusted data to trusted data take place in a controlled fashion, and appear explicit in the program. Also, the analysis should be able to provide a compile-time guarantee that all trustworthiness `check`s are met. The goal of the static analysis is not to improve run-time performance.

We distinguish two kinds of data: *trusted* data and *untrusted* data. Trusted data will typically arise from program constants, company databases, trustworthy persons, cryptographically verified input from a known partner etc. All other pieces of data, such as data obtained via an insecure network connection or from world writable files is regarded as untrusted.

The figure below is an abstract picture of the data-dependences in a typical function. We see that the result of the function depends on both the function's arguments and its environment as symbolized by the paths entering the function through the sides of the box. Suppose the result of that function is stored in a company database or used in a transaction transferring money between accounts. We clearly want to be able to trust the output of that function, regardless of how the result of the function is derived from the arguments and the environment of the function. In order to make the result trustworthy the programmer has inserted certain checks in the function, symbolized by the small circles. However, in this case there are still paths in the function such that untrustworthy information may leak through if the function is called with untrustworthy arguments. A trust analysis can warn about the existence of such unchecked paths.



Trust analysis has much in common with security flow analysis in the sense of Denning [Den82]. Security flow analysis tracks data dependences ensuring that secret data can never directly or indirectly be put in insecure places. The main difference between trust analysis and security analysis is the introduction of the `trust` construct in trust analysis that explic-

itly makes a piece of data appear trustworthy to the analysis (presumably after appropriate checks). In security analysis this would correspond to a hypothetical `declassify` construct making secret data suddenly appear non-secret. In security analysis such a construct would make no sense, and none of the security analyses have considered it, whereas the `trust` construct makes very good sense in the trust analysis setting. Likewise, the `check` construct of trust analysis would correspond to a check for unclassified data.

As in much of computer science, the early work on data security dates back to the beginning of the seventies. The basic model for security in computer systems goes back to the work by Bell and LaPadula at MITRE [BL73, BL76]. Later the seminal work by Denning and Denning in the mid-seventies [Den76, DD77, Den82] initiated the idea of applying program analysis to the computer security problem. The analyses of Denning are basically applications of abstract interpretation [CC77] to the problem of security flow analysis of Pascal-like programs, that is, a first-order imperative programming language with procedures and structured control flow. The semantic soundness of the analysis was not proved formally. Later Andrews and Reitman [AR79, AR80] gave an axiomatic re-formulation of the analysis for Concurrent Pascal, also without a formal proof of soundness.

In the beginning of the eighties the formal soundness of security analysis algorithms became more of an issue. There are roughly two ways to reason formally about security in programs. One is based on a so-called *instrumented semantics*, where the semantics associates a security classification (a tag) with every value in the program. The quest is then to make the analysis find a safe compile-time approximation of these run-time tags. The other method is based on the notion of *non-interference* by Goguen and Meseguer [GM82]. Here the programs themselves are associated with security classifications computed by the compile time analysis. A program is non-interfering if it affects no variables classified at lower security levels than the program itself, and uses no variables classified as more secure than the program itself. More concretely: a program classified secret cannot affect the value of a variable classified public, and cannot use the value of a variable classified top-secret. Using non-interference as the soundness condition avoids the need for an (artificial) instrumented semantics, at the cost of a possibly more complex proof. Note that non-interference is not well-suited as a soundness criterion for trust analysis, as a perfectly well-formed program that manipulates untrustworthy data may use the `trust` construct to legally affect trustworthy data.

Mizuno and Schmidt [Miz89, MS92] presented a security flow analysis based on abstract interpretation together with a soundness proof relative to an instrumented denotational semantics, for a first order modular imperative language. Banâtre, Bryce and Le Metayer [BBM94] derived a security flow analysis from an axiomatic semantics for a first order imperative language.

The paper [Ørb95] (Chapter 2 of this thesis) introduced the notion of trust analysis, and gave two methods for trust analysis of a first order imperative language: one based on abstract interpretation, and one based on constraint solving. Both methods were proved sound with respect to an instrumented operational semantics.

The paper [PØ95a] (Chapter 3 in this thesis) presented a trust analysis for an extended λ-calculus in the form of an annotated type system.

1.1.1 Why Trust Analysis isn't...

- **Binding-time analysis**: If trust analysis was equivalent to binding-time analysis[PS92, HM94] then one would equate trusted with static and distrusted with dynamic, and without using any of our special constructs this analogy goes a long way. However, the trust construct would correspond to an unrestricted "down-lift" operation able to convert arbitrary dynamic data to static data, something that is clearly unsound in a binding-time analysis. Our distrust construct would correspond nicely to the lift operation, but again the check construct has no counterpart in ordinary binding-time analysis. It has, however, been remarked that a check construct would be useful in a binding-time analysis, as a means to test whether the BTA finds the intended binding-time for a particular expression.

- **Dynamic typing**: Dynamic typing also known as tagging/untagging analysis [AM91, Hen92, WC94] aims to remove type tags as much as possible in a dynamically typed language. One might be tempted to view, say, distrust as a tagging operation and trust as the corresponding untagging operation. However, this does not explain how check should be interpreted and it doesn't match with our application type rule, in that applying a tagged function to an argument does not necessarily result in a tagged result.

  One idea is that trust analysis might be used as a kind of soft typing extension to languages like C or C++ which are almost strongly typed, but contain loopholes such as unrestricted type-casts. The idea is to essentially have two copies of every C type, a trusted variant and an untrusted variant, such that the compiler could guarantee no type errors for variables having a trusted type, whereas the compiler could insert run-time checks for values of untrusted types. However, it turns out that this kind of analysis is not equivalent to trust analysis, as illustrated by the following C example:

  ```
  if ((int) p)
    x = 5;
  else
    x = 7;
  ```

  Since type-casting is supposed to correspond to distrust and the two assignments to x are dependent on the condition; following [Ørb95] x would have to be treated as untrusted after the if-statement. This is not what we want for this kind of analysis, because in both cases x would clearly contain an integer. This illustrates that trust analysis does not serve the purpose of this analysis.

1.2 Dependence Analysis

A central notion in program analysis is that of *data dependence*. The information that the value of a variable depends on the value of another variable is used in many places: in binding-time analysis [PS92], in program slicing [Wei84], in security analysis [Den82], in strictness and neededness analysis [HY86, Nie87] etc.

Usually, dependences are thought of as boolean: a variable either depends on another variable or it doesn't. The dependence analysis developed in this thesis takes a more nuanced view. For example, from the assignment:

$x := y$ DIV 5

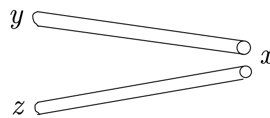
we will extract more information than merely the dependence of x upon y. From the presence of the integer division operator we can also infer that the variables x and y must hold integers at this point in the program (assuming a particular type system), and thus that the dependence between the two variables is of an integer kind.

We view dependences as existing between *slots* (usually program variables holding values), and thereby disconnected from the program text. Other notions of dependence emphasize the dependence of program statements upon variables and other program statements. In a sense, the view taken here is akin to polymorphic type systems, where the presence of a type variable in both the argument and result positions of a function type (e.g. as $(\alpha, \beta) \to \alpha$) encodes a dependence between the input and output of functions of that type without referring to the particular implementation of functions of that type.

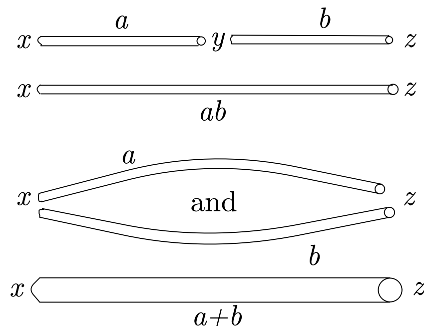
It may be useful to think of dependences as *channels* between value slots. The statement:

$x := y + z$

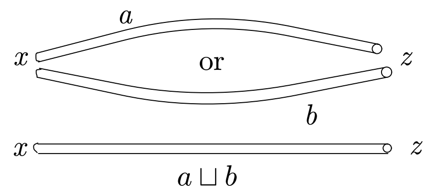
gives rise to "number" channels from y and z to x, as pictured below:



Notice that channels, not slots, are associated with types in this approach. Slots are allowed to hold values of different types at different points in time. Concatenating two channels yields a new channel between the end points, and two parallel channels between the same end points can be added to yield a new channel with the combined capacity, as illustrated below:



A third combination of channels exists to handle the case of conditional execution. The join $a \sqcup b$ of two parallel channels models that values flow either along channel a or along channel b, but not along both at once.



Considering channels between single value slots quickly becomes uninteresting, so to model programs with more than one variable, we consider vectors of slots. As the j'th output slot

may depend on the i'th input slot, we organize the dependences between vector elements in a dependence matrix. Sequential composition of programs is modeled by matrix multiplication.

In order to model dependence on constants, and handle value consumers (procedures that only consume values, such as `print`) we introduce two special slots: a source and a sink, together with augmented rules for composition. This gives rise to the notion of dependence quadruples, and an algebra of such quadruples.

A more in-depth introduction to dependence algebra can be found in Section 1.3.3, and in Chapter 4 which also relates dependence algebra-based analysis to classical frameworks such as abstract interpretation and type inference.

One of the virtues of dependence algebra-based analysis is that, by emphasizing the *connections* between slots manifest in the program, the analysis can be program-point sensitive. In type systems the underlying idea is to associate a type to each slot, such that a given slot is associated with a fixed (possibly polymorphic) type throughout the program. In the case of a polymorphic type no information about the particular type of a slot (variable) at a given point in the program is available. In contrast to this, a dependence-based analysis may allow a slot to hold different types of values at different points in the program without sacrificing detailed knowledge of the type of value in the slot at those points. This is exemplified by the trust analysis for C developed in Chapter 5, where the same variable is allowed to hold both trusted and untrusted values at different points in the program.

The work that probably comes closest to the notion of dependence analysis developed in the present thesis is the paper by Bergeretti and Carré [BC85] on information flow relations. In their paper they extract three relations from a program S: a relation μ_S between program expressions and variables, such that $e\mu_S v$ whenever e may be used in obtaining the value of v on exit from S; a relation λ_S between variables and program expressions, such that $v\lambda_S e$ whenever the value of v may be used in the evaluation of e; and finally a relation ρ_S on program variables, such that $v\rho_S v'$ whenever the value of v' on exit from S depends on the value of v on entry to S. Our notion of dependence is most closely related to their ρ_S relation.

However, Bergeretti and Carré explicitly consider *relations* between variables, represented as boolean matrices. The dependence matrices developed in Chapter 4 generalize the boolean algebra of relations to the richer dependence algebras. Also, our analysis gives results independent of program syntax, as we have no direct relation between variables and concrete program expressions or program points.

1.3 Overview of the Thesis

The following sections give an informal summary of the individual chapters of the thesis. Chapters 2 and 3 are almost verbatim copies of published papers, and the notation used in those chapters therefore differs slightly from the notation used elsewhere in the thesis.

1.3.1 Simple Trust Analysis

Chapter 2 describes a trust analysis for a simple, strongly typed, `while` language without procedures, but including pointers. Two methods for trust analysis are given: the first based on abstract interpretation, and the second based on constraint generation. The analysis based on abstract interpretation is more precise than the constraint based analysis, but requires whole programs to be analyzed at once. The constraint based analysis allows for separate generation of constraints for individual program fragments, and is thereby more modular. It is

also shown to have a lower time complexity than the analysis based on abstract interpretation. Both analyses deal explicitly with pointers and aliasing problems.

The abstract interpretation analysis is proved sound with respect to an instrumented semantics by fairly standard means, whereas the constraint based analysis is related via a safety result to the abstract interpretation, and thereby indirectly to the instrumented semantics. This also provides a direct comparison of the two methods of analysis.

1.3.2 Higher-Order Trust Analysis

Chapter 3 describes a trust analysis for a higher order functional language in terms of an annotated type system building on the simply typed λ-calculus.

First an extension of the λ-calculus with the constructs trust, distrust, and check is presented in the form of a reduction system. The reduction system is proved Church-Rosser. A denotational semantics for the calculus is also given and related to the reduction system by a semantic soundness theorem.

The trust analysis is formulated in terms of an annotated type system, which is shown to have the subject reduction property with respect to the reduction rules. The type system is related to the standard simply-typed λ-calculus, and two simulation results are obtained. Via the simulation results strong normalization for well-typed terms of the extended calculus is proved. Finally, a cubic time type inference algorithm building on constraint generation is presented and proved sound and complete with respect to the type system. The notion of trust analysis for λ-calculus is compared to other analyses such as binding-time analysis and dynamic typing.

1.3.3 Dependence Algebra

Chapter 4 develops the theory of abstract dependence algebra. First the technical notions of slots and dependence are motivated and fixed. The axioms for a dependence algebra (written **DAlg**) are given and motivated, then the computation of fixed points of functions over dependence algebras is discussed, as the dependence algebra ordering may in general lack a least element. The lack of a bottom element invalidates the usual iterative method of finding fixed points, but a simple solution to the problem is provided.

Vectors and matrices over dependence algebras are then introduced, and the set of $n \times n$ matrices over a **DAlg** is shown to again form a dependence algebra. Then the notion of dependence quadruples (quads), akin to affine maps, is introduced, operations of composition, sum and join on quads are defined, and some algebraic laws of the operations on quads are shown.

Next, the important notion of the *image* of a vector through a quad is defined, and an important relation between image and quad composition is shown. The dual notion of pre-image is also defined. The chapter then goes on to define concatenation of vectors, matrices and quads, and relate this operation to the image and composition operations. In some analyses, the quads used may grow very large, so a method for constructing small approximate representations of large quads is described.

The concept of abstracting a program into a single quad is powerful enough for the trust analysis of C programs in Chapter 5, but it is not enough for the analysis of action semantics in Chapter 6. We therefore consider sets of quads, and operations on these sets. The operations of composition, concatenation, sum and join are extended in a natural way to sets of quads.

The quad sets needed in Chapter 6 will in general be infinite, so we need finite approximations of such sets to implement analyses based on the framework. Such a finite approximation is also developed in Chapter 4 in the form of grids. Operations of composition, concatenation, sum and join are defined on these grids and shown to be sound approximations of the corresponding operations on sets of quads.

The last sections of Chapter 4 describe the construction of a particular non-trivial **DAlg**-structure from a distributive lattice, and define the notion of a reverse image of a vector through a quad. An important relation between quad composition and the reverse image is shown. Finally, connections to abstract interpretation and type inference are discussed.

1.3.4 Trust Analysis of C Programs

Chapter 5 describes a practical implementation of a trust analysis for the C programming language. The implementation is based on the dependence algebra theory developed in Chapter 4, and thus the two pillars of the thesis are combined.

The chapter first outlines the structure of the implementation. Pragmatic issues of parsing and type checking C code are briefly discussed, and a few syntactic extensions for the specification of trust information for external functions are described.

After a program is parsed into a syntax tree, it is compiled into a non-standard control flow graph (CFG) representation. The compiler incorporates a peephole optimizer to improve the efficiency of the ensuing analysis. A formal abstract semantics of the CFG language is given in terms of a transition system, describing a stack machine. An analysis of implementation dependent evaluation order is described briefly.

The notion of control dependence captures the influence of a conditional on the following code. In a structured representation the control influence of a conditional is trivially confined to the syntactic branches of the conditional, but in an unstructured representation such as a control flow graph with arbitrary `goto`s it takes more work to find the extent of a control influence. An algorithm for decorating a CFG with transitive control dependences is given and proved correct with respect to the standard notion of control dependence.

Next, the trust analysis on the level of the CFG representation is given in terms of a transition system, and proven sound with respect to the operational semantics of the CFG language. The soundness proof is factored into soundness properties for single statements, for sequences, and for the inter-procedural aspect involving calls and returns. The analysis is also shown to be separable, that is, individual functions can be analyzed separately.

Finally, some remarks on the actual implementation are given, and timing measurements on a few small, but realistic, codes are given, indicating the practical usefulness of the implementation. A discussion of the applicability of the analysis to existing programs concludes.

1.3.5 Soft Type Inference for Action Semantics

Chapter 6 builds on the theory of dependence quads to present a soft type inference system for action semantics specifications.

The action semantics formalism invented by Mosses [Mos92] consists of an algebraic framework for specifying data and a dynamically typed language called *action notation* used to specify the dynamic semantics of programming languages. Section 6.2 gives a short introduction to the specification formalism.

The "soft" in soft type inference means that the type system is forgiving: no set of semantic equations is deemed type incorrect, the type inference just tries to infer as much information from the equations as possible.

The inferred type information is intended to help the writer of the semantics avoid simple errors by providing information about the semantic equations before they are applied to a particular program.

The analysis computes a set of dependence quads for each semantic function in a specification, and type information about the actions generated from a semantic function can be gleaned from this set of quads.

A subset of action notation incorporating the basic and functional facets including higher order functions is defined, and a natural, big step operational semantics for the subset is given.

Subsequently, a simplification algorithm for action semantic equations is defined. The algorithm removes the unfolding and unfold constructs, and names all abstractions, in order to explicitly label all kinds of recursion.

The simplified semantic equations are then transformed into an intermediate representation language with much fewer constructs than the syntax of action notation.

Finally, the set of equations expressed in the intermediate language is analyzed. The analysis combines the computation of (finite approximations of) sets of quads with a control flow analysis akin to the 0-CFA analysis of Shivers [Shi88, Shi91] to approximate the higher-order control flow. A formal specification of the analysis algorithm in terms of an inference system is given.

The analysis algorithm is then proved sound with respect to the operational semantics. The soundness theorem also describes how to extract type information from the sets of dependence quads computed by the analysis.

Finally, Chapter 7 concludes and summarizes the results presented in the present thesis.

1.4 Acknowledgments

Chapter 2 is based on the paper [Ørb95] that appears in the proceedings of the TAPSOFT/FASE'95 conference. Chapter 3 is joint work with Jens Palsberg. An extended abstract of the chapter appears in the conference proceedings of SAS'95 [PØ95a], and a full version of the paper is to appear in *Journal of Functional Programming*.

Chapter 2

Can you Trust your Data?

2.1 Introduction

This chapter discusses a static program analysis that can be used to check that the validity of data is only promoted to higher levels of trust in a conscious and controlled fashion.

It is important to stress that the purpose of the analyses is *not* to improve run-time performance, but to give warnings to the programmer whenever untrustworthy data are being unduly trusted.

In the rest of the chapter we try to motivate the need for a trust analysis. We give an instrumented semantics for a simple first order language with pointers, in effect keeping track of the trustworthiness of data at run-time. Then an abstract interpretation is presented, approximating the analysis statically. Finally, in order to gain separate analysis of separate program modules as well as better time complexity, a constraint based analysis is presented. The constraint based analysis is proved to be a safe approximation of the abstract interpretation.

2.2 Motivation

Many computer systems handle information of various levels of trustworthiness. Whereas the contents of the company database can usually be trusted, the input gathered via a modem, or from a part-time secretary may not be trusted as much, and data validation and authentication routines must ensure the validity of data before it is promoted to a higher level of trust and entered into the database.

That there is a need for some method to control the propagation of trust in real-life computer programs is witnessed for example by the security hole recently found in the Unix `sendmail` program [CER94]. `Sendmail` is the mail forwarding program running on the majority of Unix machines on the Internet. The security hole allowed one to give the program a certain devious input (in an e-mail message) that would result in having arbitrary commands executed on the machine with superuser privileges. Had an analysis like the one described in this chapter been run on the `sendmail` sources it is likely that such a breach in security could have been noticed in advance. See below.

As an example of the kind of analysis envisioned, Perl [WS91] implements "taint" checks at run-time to help ensure that untrustworthy values are not put in places (such as a process' user-id) where only trusted data should go. This "tainting" is very closely related to the

instrumented semantics given below.

We aim at finding a *static* program analysis, i.e. an analysis run only once when a program is compiled, such that the programmer is warned if and when data is promoted from untrustworthy to trustworthy in an uncontrolled fashion. Clearly there will be a need to promote data from untrusted to trusted, but with the envisioned analysis we can guarantee that the promotion takes place in an explicit and conscious way.

In [Den76, DD77] Denning and Denning present a flow analysis for what they call "secure information flow". Their analysis in a sense attacks the dual of the problem attacked in this chapter. Their aim is to prevent privileged information from leaking out of a trusted computer system, whereas "trust analysis" aims at preventing untrustworthy information from entering into a trusted computer system.

2.2.1 The Sendmail example

Inside the `sendmail` C code there is a routine, `deliver()`, that delivers an e-mail message to an address:

```
void deliver(MSG m, ADR a, ...) {
  ...
  setuid(a.uid);
  ...
}
```

For some addresses, the `uid` field makes no sense and is uninitialized. In current sources, the `ADR` structure contains a bit that should be set just when the `uid` field is valid, and this bit is tested in several places at run-time before the `uid` field is used. The security hole existed because the programmer had forgotten to insert enough of these checks and consequently, under certain circumstances one was able to circumvent the checks and gain superuser privileges.

With a trust analysis, a reasonable choice is to make the `setuid()` system-call accept only trusted values, as it sets the user-id of the current process. This forces `a.uid` to be a trusted value for compilation of `deliver()` to succeed. One would then have just *one* place, namely in a validation procedure, where the value of an address' `uid` field is promoted to trusted.

```
ADR validate_address(ADR a) {
  ADR a1;
  ... some validation, fill in appropriate parts of a1.
  ... we may now trust the contents of a.uid.
  a1.uid = trust(a.uid);
  return a1;
}
```

The trust analyzer will now be able to ensure the programmer that only trusted values are passed to `setuid()`. And all the run-time checks on the validity bit are no longer needed as the trust checks are wholly static.

2.3 The While language

Since a large part of security conscious programs today are written in C, a stripped down imperative C-like language with pointers is explored. The abstract syntax for the language is defined by the following BNF:

$$
\begin{array}{rcl}
I & ::= & \textit{variable names} \\
P & ::= & \mathsf{deref}\ P \mid I \\
E & ::= & P \mid E + E \mid \ldots \mid \textit{const} \mid \mathsf{addr}\ I \mid \mathsf{trust}\ E \mid \mathsf{distrust}\ E \\
S & ::= & \mathsf{while}\ E\ \mathsf{do}\ S \mid S; S \mid P\ :=\ E
\end{array}
$$

Informally, I denotes identifiers, P denotes pointer expressions, E denotes arithmetic and boolean expressions and S denotes statements. Initially the language included first order procedures, but due to lack of space and since they can be added on in a straightforward way they have been left out. How to do this is briefly discussed in Section 2.8.

We assume programs are *strongly typed* (i.e. like in Pascal), but leave out type declarations such as int or bool as the only thing that matters for our purpose is whether a variable contains a pointer or a scalar (non-pointer) value.

Deref dereferences pointers. If-statements can be emulated by while loops. This saves a syntactic construct.

Notation: The following conventions are used for meta-syntactic variables: i ranges over identifiers I; e, e_1 and e_2 range over expressions E; p ranges over pointer expressions P and s, s_1 and s_2 range over statements S.

2.4 Standard Semantics

The standard semantics for the language is given in a sugared direct style. In order to conserve space we give only the *dynamic* semantics for statements.

Below are the definitions of the semantic domains. *Addr* is the set of possible addresses in memory. The set of possible program values, *Val*, includes at least integers, booleans and addresses.

Environments (*Env*) map identifiers to addresses, and memories (*Mem*) map addresses to values. Note that environments are assumed to be *injective*.

The semantic function $\mathcal{E}$ gives meaning to side-effect free expressions and $\mathcal{S}$ gives meaning to statements. *Notation:* We use $\uplus$ for disjoint set union.

$$
\begin{array}{rcl}
\textit{Addr} & \subseteq & \mathbb{N} \\
\textit{Val} & \supseteq & \textit{Int} \uplus \textit{Bool} \uplus \textit{Addr} \\
\textit{Env} & = & I \to \textit{Addr} \\
\textit{Mem} & = & \textit{Addr} \to \textit{Val} \\
\mathcal{E} & : & E \to \textit{Env} \to \textit{Mem} \to \textit{Val} \\
\mathcal{S} & : & S \to \textit{Env} \to \textit{Mem} \to \textit{Mem} \\
\textit{addr} & : & P \to \textit{Env} \to \textit{Mem} \to \textit{Addr}
\end{array}
$$

$$\begin{array}{rcl} M & \in & \mathit{Mem} \\ A & \in & \mathit{Env} \end{array}$$

$$\begin{array}{rcl}
\mathcal{E}\ i\ A\ M & = & M(A(i)) \\
\mathcal{E}\ [\![\mathsf{addr}\ i]\!]\ A\ M & = & A(i) \\
\mathcal{E}\ [\![\mathsf{deref}\ p]\!]\ A\ M & = & M(\mathcal{E}\ p\ A\ M) \\
\mathcal{E}\ [\![e_1 + e_2]\!]\ A\ M & = & (\mathcal{E}\ e_1\ A\ M) + (\mathcal{E}\ e_2\ A\ M) \\
\mathcal{E}\ [\![\mathsf{trust}\ e]\!]\ A\ M & = & \mathcal{E}\ e\ A\ M \\
\mathcal{E}\ [\![\mathsf{distrust}\ e]\!]\ A\ M & = & \mathcal{E}\ e\ A\ M \\
\mathcal{E}\ \mathit{const}\ A\ M & = & \mathit{const}
\end{array}$$

By strong typing we can assume that trust _ is applied to scalar values only. This will be important for the constraint generation analysis. *Notation:* The memory $M[v/a]$ is as M except that the address a is mapped to the value v, and similarly for environments.

$$\begin{array}{rcl}
\mathit{addr}\ i\ A\ M & = & A(i) \\
\mathit{addr}\ [\![\mathsf{deref}\ p]\!]\ A\ M & = & M(\mathit{addr}\ p\ A\ M) \\
\\
\mathcal{S}\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A\ M & = & \mathsf{let}\ b = \mathcal{E}\ e\ A\ M\ \mathsf{in} \\
 & & \quad \mathsf{if}\ b\ \mathsf{then}\ \mathcal{S}\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A\ (\mathcal{S}\ s\ A\ M) \\
 & & \quad \mathsf{else}\ M \\
\mathcal{S}\ [\![p\ :=\ e]\!]\ A\ M & = & M[\mathcal{E}\ e\ A\ M/(\mathit{addr}\ p\ A\ M)] \\
\mathcal{S}\ [\![s_1; s_2]\!]\ A\ M & = & \mathcal{S}\ s_2\ A\ (\mathcal{S}\ s_1\ A\ M)
\end{array}$$

2.5 Instrumented Semantics

In order to keep track the trustworthiness of values at run-time, we give an instrumented semantics that associate each value with a flag telling whether the value can be trusted or not. This is to be taken as the *definition* of the desired analysis.

$$\begin{array}{rcl}
\mathit{Tr} & = & \{\bot, \top\} \\
\mathit{Val}_I & = & \mathit{Val} \times \mathit{Tr} \\
\mathit{Mem}_I & = & \mathit{Addr} \rightarrow \mathit{Val}_I \\
\mathcal{E}_I & : & E \rightarrow \mathit{Env} \rightarrow \mathit{Mem}_I \rightarrow \mathit{Val}_I \\
\mathcal{S}_I & : & S \rightarrow \mathit{Env} \rightarrow \mathit{Mem}_I \rightarrow \mathit{Tr} \rightarrow \mathit{Mem}_I \\
\mathit{addr}_I & : & P \rightarrow \mathit{Env} \rightarrow \mathit{Mem}_I \rightarrow \mathit{Addr} \\
M_I & \in & \mathit{Mem}_I
\end{array}$$

We equip the set Tr with a total ordering ($\leq$) such that $\bot \leq \top$ in order to make it a lattice. The least upper bound operation on this lattice will be denoted by $\vee$, which will

also be used to denote the *lub* of environments by point-wise extension. The idea is that $\bot$ corresponds to trusted data, and $\top$ corresponds to untrusted data. *Notation:* $\langle\cdot,\cdot\rangle$ forms Cartesian products and π_n is the n'th projection. t ranges over Tr and v over Val.

$$
\begin{array}{rcl}
\mathcal{E}_I\ i\ A\ M_I & = & M_I(A(i)) \\
\mathcal{E}_I\ [\![\mathsf{addr}\ i]\!]\ A\ M_I & = & \langle A(i),\ \bot\rangle \\
\mathcal{E}_I\ [\![\mathsf{deref}\ p]\!]\ A\ M_I & = & \mathsf{let}\ \langle v,t\rangle = \mathcal{E}_I\ p\ A\ M_I\ \mathsf{in} \\
 & & \quad \langle \pi_1(M_I(v)),\ t \vee \pi_2(M_I(v))\rangle \\
\mathcal{E}_I\ [\![e_1 + e_2]\!]\ A\ M_I & = & (\mathcal{E}_I\ e_1\ A\ M_I)\hat{+}(\mathcal{E}_I\ e_2\ A\ M_I) \\
\mathcal{E}_I\ [\![\mathsf{trust}\ e]\!]\ A\ M_I & = & \langle \pi_1(\mathcal{E}_I\ e\ A\ M_I),\ \bot\rangle \\
\mathcal{E}_I\ [\![\mathsf{distrust}\ e]\!]\ A\ M_I & = & \langle \pi_1(\mathcal{E}_I\ e\ A\ M_I),\ \top\rangle \\
\mathcal{E}_I\ const\ A\ M_I & = & \langle const,\ \bot\rangle \\
\langle v_1, t_1\rangle \hat{+} \langle v_2, t_2\rangle & = & \langle v_1 + v_2, t_1 \vee t_2\rangle
\end{array}
$$

The last parameter to $\mathcal{S}_I$ is used in connection with while loops, the reason being that if the condition in the loop cannot be trusted, then all variables assigned in the loop can no longer be trusted as they may depend on the number of iterations taken. This is also known as implicit information flow in the various security flow analyses.

$$
\begin{array}{rcl}
addr_I\ i\ A\ M_I & = & A(i) \\
addr_I\ [\![\mathsf{deref}\ p]\!]\ A\ M_I & = & \pi_1(M_I(addr_I\ p\ A\ M_I))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}_I\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A\ M_I\ t & = & \mathsf{let}\ \langle v,t'\rangle = \mathcal{E}_I\ e\ A\ M_I\ \mathsf{in} \\
 & & \quad \mathsf{if}\ v\ \mathsf{then} \\
 & & \quad\quad \mathcal{S}_I\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A\ (\mathcal{S}_I\ s\ A\ M_I\ (t \vee t'))\ t \\
 & & \quad \mathsf{else}\ M_I \\
\mathcal{S}_I\ [\![p\ :=\ e]\!]\ A\ M_I\ t & = & \mathsf{let}\ \langle v,t'\rangle = \mathcal{E}_I\ e\ A\ M_I\ \mathsf{in} \\
 & & \quad M_I[\langle v,\ t \vee t'\rangle/(addr_I\ p\ A\ M_I)] \\
\mathcal{S}_I\ [\![s_1; s_2]\!]\ A\ M_I\ t & = & \mathcal{S}_I\ s_2\ A\ (\mathcal{S}_I\ s_1\ A\ M_I\ t)\ t
\end{array}
$$

If we define the relation $\cong$ between real and instrumented memories as

$$M \cong M_I \iff \mathrm{dom}(M) = \mathrm{dom}(M_I) \text{ and } M_I(x) = \langle M(x), t\rangle, \text{ for some } t,$$

we can state the relationship between the standard semantics and the instrumented semantics as follows:

Proposition 2.1 (Faithfulness) *The instrumented semantics faithfully executes the program according to the standard semantics. Formally, if $M \cong M_I$ then:*

$$\mathcal{S}\ s\ A\ M = M' \quad \Rightarrow \quad \exists M'_I : \mathcal{S}_I\ s\ A\ M_I = M'_I$$

and

$$\mathcal{S}_I\ s\ A\ M_I = M'_I \quad \Rightarrow \quad \exists M' : \mathcal{S}\ s\ A\ M = M',$$

and in both cases $M' \cong M'_I$.

Proof. Each way is proved by induction in the number of applications of the semantic function. The semantics emulate each other step by step. □

2.6 Abstract Interpretation

The instrumented semantics has the drawback that it propagates the trust of variables only at run-time. Below is presented an abstract interpretation [CC77] of the language computing an approximation to the trust tags and not the actual values.

Since the actual values are not known during the abstract interpretation neither are the addresses, hence environments and memories are collapsed into abstract environments mapping identifiers directly to "trust signatures". *Notation:* 2^I denotes the set of subsets of I.

$$
\begin{array}{rcl}
Val_A & = & Tr \cup 2^I \\
Env_A & = & I \rightarrow Val_A \\
\mathcal{E}_A & : & E \rightarrow Env_A \rightarrow Val_A \\
\mathcal{S}_A & : & S \rightarrow Env_A \rightarrow Tr \rightarrow Env_A \\
addr_A & : & P \rightarrow Env_A \rightarrow 2^I \\
asg & : & Val_A \rightarrow Env_A \rightarrow Val_A \rightarrow Env_A \\
M_A & \in & Mem_I \\
v & \in & Val_A \\
A_A & \in & Env_A
\end{array}
$$

We extend the total ordering on Tr to a partial ordering on Val_A such that

$$\forall v \in Val_A : \bot \leq v \leq \top \text{ and } a, b \in 2^I \Rightarrow (a \leq b \iff a \subseteq b).$$

This makes Val_A a complete lattice, and for any finite collection of programs, finite as well. $\vee$ is used for least upper bound on this lattice as well. The ordering is extended pointwise to abstract environments.

The idea is that $\bot$ corresponds to trusted scalars. A set of identifiers corresponds to a trusted pointer that may point to any of the variables mentioned in the set. $\top$ corresponds to untrusted values of any kind. Letting abstract environments map identifiers to sets of identifiers, instead of keeping both information about the pointer and the data pointed to in the abstract environment, is done to handle pointer aliasing. *Notation:* For brevity, define $A_A(\top) = \top$, and for $a \subseteq I$ let $A_A(a) = \bigvee_{i \in a} A_A(i)$.

$$
\begin{array}{rcl}
\mathcal{E}_A\ i\ A_A & = & A_A(i) \\
\mathcal{E}_A\ [\![\mathsf{addr}\ i]\!]\ A_A & = & \{i\} \\
\mathcal{E}_A\ [\![\mathsf{deref}\ p]\!]\ A_A & = & \bigvee A_A(\mathcal{E}_A\ p\ A_A) \\
\mathcal{E}_A\ [\![e_1 + e_2]\!]\ A_A & = & (\mathcal{E}_A\ e_1\ A_A) \vee (\mathcal{E}_A\ e_2\ A_A) \\
\mathcal{E}_A\ [\![\mathsf{trust}\ e]\!]\ A_A & = & \bot \\
\mathcal{E}_A\ [\![\mathsf{distrust}\ e]\!]\ A_A & = & \top \\
\mathcal{E}_A\ const\ A_A & = & \bot
\end{array}
$$

The auxiliary *asg* function monotonically assigns a new trust value to a set of identifiers in an abstract environment. $addr_A\ p\ A_A$ yields the set of variables that might be assigned to when p is the left hand side of an assignment. *Notation:* $\text{dom}(M)$ denotes the domain of the map M.

$$\begin{aligned}
asg\ t\ A_A\ \top &= \{(i \mapsto \top) \mid i \in \text{dom}(A_A)\} \\
asg\ t\ A_A\ s &= \{(i \mapsto A_A(i) \vee t) \mid i \in s\} \\
&\quad \cup\{(i \mapsto A_A(i)) \mid i \in \text{dom}(A_A) \setminus s\} \\
addr_A\ i\ A_A &= \{i\} \\
addr_A\ [\![\mathsf{deref}\ p]\!]\ A_A &= \bigvee A_A(addr_A\ p\ A_A)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}_A\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A_A\ t &= \mathsf{let}\ A'_A = \mathcal{S}_A\ s\ A_A\ (t \vee \mathcal{E}_A\ e\ A_A) \\
&\quad\ \mathsf{in\ if}\ A'_A \leq A_A\ \mathsf{then}\ A_A\ \mathsf{else}\ \mathcal{S}_A\ [\![\mathsf{while}\ e\ \mathsf{do}\ s]\!]\ A'_A\ t \\
\mathcal{S}_A\ [\![p\ := \ e]\!]\ A_A\ t &= asg\ (\mathcal{E}_A\ e\ A_A \vee t)\ A_A\ (addr_A\ p\ A_A) \\
\mathcal{S}_A\ [\![s_1; s_2]\!]\ A_A\ t &= \mathcal{S}_A\ s_2\ (\mathcal{S}_A\ s_1\ A_A\ t)\ t
\end{aligned}$$

To relate the instrumented and abstract semantics an ordering between instrumented and abstract values is defined relative to an environment:

$$A \vdash \langle v, t\rangle \sqsubseteq a$$

if and only if $a = \bot \Rightarrow t = \bot$ and $a \subseteq I \Rightarrow (t = \bot\ \textit{and}\ v \in A(a))$.

Informally, the first implication means that if the abstract semantics says that a value is a trustworthy scalar then indeed it is marked trusted in the instrumented semantics. The second implication means that if the abstract semantics thinks a value is a pointer to one of the variables in a set a then by the instrumented semantics the value is indeed trustworthy and is a pointer to one of the variables in the set a.

The relation is extended to relate combined instrumented memories and environments with abstract environments like this:

$$M_I \circ A \sqsubseteq A_A$$

if and only if $\text{dom}(M_I \circ A) = \text{dom}(A_A)$ and $A \vdash (M_I \circ A)(i) \sqsubseteq A_A(i)$ for all variables $i \in \text{dom}(A_A)$

We relate the abstract interpretation to the instrumented semantics in the following way:

Proposition 2.2 (Safety) *If a statement is executed in an environment A and a memory M_I by the instrumented semantics, and the abstract environment A_A is a safe approximation of A and M_I then the result of the abstract interpretation is a safe approximation of the memory resulting from the instrumented semantics. Formally: If*

$$\mathcal{S}_I\ s\ A\ M_I\ t\ =\ M',\ \ M_I \circ A \sqsubseteq A_A,\ \ \mathcal{S}_A\ s\ A_A\ t_A\ =\ A'\ \ \textit{and}\ \ t \leq t_A$$

then $M' \circ A \sqsubseteq A'$.

Proof. See Section 2.10.

The abstract interpretation terminates. It is clear that $\mathcal{E}_A$ terminates as it is defined inductively in the (finite) structure of expressions, and no fixpoints are computed. The only possibility for $\mathcal{S}_A$ to diverge would be in the while case where a fixpoint is computed, but by Lemma 2.9 the fixpoint is computed of a monotone function over a lattice of finite height, hence the fixpoint can be found in finite time by iteration.

If we let n denote the number of distinct variables used in a program, let l denote the number of statements and expressions, and let m denote the greatest depth of while-loop nests in the program, the number of least upper bound operations on Val_A executed by the abstract interpretation will be in $O((n+l)^{2m})$. In the worst case, the least upper bound operation on Val_A can be computed in $O(n)$ time. This sounds worse than it really is. For ordinary programs m will be a small constant, and the complexity of analyzing a while-loop is at most $O(n_b^2)$ times the complexity of analyzing the loop body. Here n_b is the number of *pointer* variables occurring in the body of the loop.

If procedures are added to the language, fixpoints need to be computed for each procedure call, hence the time complexity will be even worse in that case.

Apart from the time complexity, the main drawback of the abstract interpretation analysis is that it needs the world to be closed; that is, the analysis cannot be run for each program module separately. In the next section a separable constraint based analysis is presented.

2.7 Constraint Generation

- Or else, what follows?
- Bloody constraint!...

William Shakespeare: Henry V, Act II, Scene 4.

The constraint generator is going to associate three constraint variables to each program variable. A solution to the generated set of constraints will assign an appropriate trust value for the program variable to one of these constraint variables.

The constraint analysis constructs constraints from any sequence of statements. This is more general than simply allowing for separate analysis of individual functions, since any sequence of statements can be (partly) analyzed out of context. This might for example be useful with an advanced module system like the Beta fragment system [KLMM93].

For the purpose of this chapter, a program consists of a top fragment that includes zero or more fragments which may again include smaller fragments and so on. The inclusion ordering of the fragments form a directed acyclic graph (DAG), as a single fragment may be included more than once, but we disallow circular dependences.

Fragments are supposed to be analyzed in a bottom-up fashion, first analyzing the leaf fragments that include no other fragments, then analyzing fragments that include only leaf fragments and so on. In effect, the fragments are treated in reverse topological order.

The domains used in the definition of the constraint generation analysis are defined below:

$$\begin{array}{rcl} V & ::= & I \mid \nabla I \mid \Delta I \\ \delta, \eta & : & V \to V \\ N & : & P \to V \end{array}$$

$$\begin{array}{rcl} Ct & = & (V \cup Tr) \times V \\ C & = & 2^{Ct} \\ \mathcal{E}_S & : & E \rightarrow C \times V \\ \mathcal{S}_S & : & S \rightarrow C \times 2^V \\ G & \in & V \end{array}$$

V is the set of constraint variables. For an identifier i, Δi, and ∇i are simply constraint variables. The intuition is that whereas i will hold the trustworthiness of the value of the program variable i, Δi will hold the trust of all the values reachable by dereferencing i any number of times. Constraint variables ∇i are used to hold the trust of addr terms.

G is a special constraint variable corresponding to the global trustworthiness of a memory. That is, if a value is assigned to the target of an untrusted pointer then that value could end up anywhere, and the trustworthiness of the entire memory is corrupted.

The pair $\langle s, t\rangle \in Ct$ codes the constraint $s \leq t$. For readability we write $\{s \leq t\}$ for such a constraint and $\{s = t\}$ as an abbreviation for $\{s \leq t, t \leq s\}$. The generated constraints will be of the form $\{\textit{variable or constant} \leq \textit{variable}\}$ over the two element lattice $\{\bot, \top\}$, hence they can be solved by simple constraint propagation in linear time. The existence of a solution is guaranteed since assigning $\top$ to all constraint variables will satisfy the generated constraints.

We assume that any set of constraints include the constraints $\{\nabla i \leq i \leq \Delta i\}$ for all identifiers i.

The function δ on V "dereferences" constraint variables:

$$\begin{array}{rcl} \delta\ \nabla i & = & i \\ \delta\ i & = & \Delta i \\ \delta\ \Delta i & = & \Delta i \end{array}$$

One may think of δ as a syntactic closure operator as $x \leq \delta x$, and $\delta x = \delta\delta x$. The function η "safely" takes the address of a constraint variable. The inequality $x \leq \eta\delta x$ can be seen to hold by inspection.

$$\begin{array}{rcl} \eta\ \nabla i & = & \nabla i \\ \eta\ i & = & \nabla i \\ \eta\ \Delta i & = & \Delta i \end{array}$$

The map N generates constraint variables from pointer expressions P:

$$\begin{array}{rcl} N\,i & = & i \\ N[\![\mathsf{deref}\ p]\!] & = & \delta N(p) \end{array}$$

$\mathcal{E}_S$ generates constraints for expressions together with the variable corresponding to the given expression. In each case n denotes a freshly created constraint variable.

$$
\begin{array}{rcl}
\mathcal{E}_S\ i & = & \langle \emptyset, i \rangle \\
\mathcal{E}_S[\![\textsf{addr}\ i]\!] & = & \langle \emptyset, \nabla i \rangle \\
\mathcal{E}_S[\![\textsf{deref}\ p]\!] & = & \textsf{let}\ \langle c, v \rangle = \mathcal{E}_S\ p \\
 & & \textsf{in}\ \langle c, \delta v \rangle \\
\mathcal{E}_S\ [\![e_1 + e_2]\!] & = & \textsf{let}\ \langle c_1, v_1 \rangle = \mathcal{E}_S\ e_1 \\
 & & \quad \langle c_2, v_2 \rangle = \mathcal{E}_S\ e_2 \\
 & & \textsf{in}\ \langle c_1 \cup c_2 \cup \{v_1 \leq n,\ v_2 \leq n\},\ n \rangle \\
\mathcal{E}_S[\![\textsf{trust}\ e]\!] & = & \langle \emptyset, n \rangle \\
\mathcal{E}_S[\![\textsf{distrust}\ e]\!] & = & \langle \{\top \leq n\}, n \rangle \\
\mathcal{E}_S\ \mathit{const} & = & \langle \emptyset, n \rangle
\end{array}
$$

$\mathcal{S}_S$ generates constraints for statements. The second part of the result is the set of constraint variables corresponding to variables assigned to within the statement. This is used to generate additional constraints for while-loops such that variables assigned to in the loop body are trusted only if the condition of the loop is.

$$
\begin{array}{rcl}
\mathcal{S}_S[\![\textsf{while}\ e\ \textsf{do}\ s]\!] & = & \textsf{let}\ \langle c_e, v \rangle = \mathcal{E}_S\ e \\
 & & \quad \langle c_s, a \rangle = \mathcal{S}_S\ s \\
 & & \textsf{in}\ \langle c_s \cup c_e \cup \{v \leq x \mid x \in a\},\ a \rangle \\
\mathcal{S}_S[\![p\ :=\ e]\!] & = & \textsf{let}\ \langle c_e, v \rangle = \mathcal{E}_S\ e \\
 & & \textsf{in}\ \langle c_e \cup \{v \leq N(p),\ \delta N(p) = \delta v, \eta N(p) \leq G\},\ \{N(p)\} \rangle \\
\mathcal{S}_S[\![s_1; s_2]\!] & = & \textsf{let}\ \langle c_1, a_1 \rangle = \mathcal{S}_S\ s_1 \\
 & & \quad \langle c_2, a_2 \rangle = \mathcal{S}_S\ s_2 \\
 & & \textsf{in}\ \langle c_1 \cup c_2,\ a_1 \cup a_2 \rangle
\end{array}
$$

A solution to the generated constraints (called a *model*) is a map m giving values to the constraint variables such that the constraints c are fulfilled, this is written $m \models c$. Formally: $m \models c$ if and only if

$$\forall \langle s, t \rangle \in c :\ m(s) \leq m(t).$$

It is clear that if $m \models c_1 \cup c_2$ then $m \models c_1$ and $m \models c_2$.

We will consider only a subset of all possible models for a set of constraints, namely so-called *coherent* models. A model m is coherent if it satisfies

$$m(a) \leq m(b) \quad \Rightarrow \quad m(\delta a) \leq m(\delta b).$$

It is clear that the model that assigns $\top$ to all variables is a coherent model, hence the existence of a coherent model is assured.

Coherent models and abstract environments can be related to each other in the following way: We write $A_A \sqsubseteq m$ if and only if

$$A_A(i) = \top \quad \Rightarrow \quad m(i) = \top$$

and

$$a \in A_A(i) \quad \Rightarrow \quad m(\Delta i) = m(a),$$

or, alternatively $m(G) = \top$.

An intuitive view of the above is that in order for a model to be a safe approximation of an abstract environment, it must assign conservative trust-values to all variables, and if a pointer p can point to a number of variables then the constraint variable Δp must be equated to the trust-values of all these variables.

The constraint generation analysis is related to the abstract interpretation by the following safety statement:

Proposition 2.3 *If*

$$\begin{array}{rcl} \langle c, v\rangle & = & \mathcal{S}_S\ s, \\ m & \models & c, \textit{ and } m \textit{ is coherent} \\ A_A & \sqsubseteq & m, \\ \forall x \in v & : & t \leq m(x), \\ A'_A & = & \mathcal{S}_A\ s\ A_A\ t \end{array}$$

then $A'_A \sqsubseteq m$.

Proof. See Section 2.11.

The constraint generation analysis is strictly weaker than the abstract interpretation in the sense that more variables are treated as untrusted, as is demonstrated by the following example:

Program	New constraint
p := addr j	$\{\nabla j \leq p,\ \Delta p = j\}$
p := addr i	$\{\nabla i \leq p,\ \Delta p = i\}$
i := distrust 8	$\{\top \leq i\}$
k := deref p	$\{\Delta p \leq k,\ \Delta k = \Delta p\}$

Remember that the following constraint is implicitly assumed: $\{p \leq \Delta p\}$. In the abstract interpretation, only i will be marked untrusted at the end, whereas in the constraint analysis the trust of i and j are linked by equality since p may point to both[1].

Generating the constraints for a program of size n takes $O(n^2)$ time in the worst case assuming that the addition of a single constraint can be done in constant time. The constraints, being of such simple nature, may be solved by value propagation in linear time in the number of constraints. All in all constraint generation and (partial) solving can be done in quadratic time in the size of the program fragment.

[1]As remarked by one of the referees, it might be possible to detect some of these situations as p is *dead* after the first assignment, so one might remove the constraints added in the first line from the final constraints and thereby get a better solution. This effect might also be achieved by removing assignments to dead variables before trust analysis.

2.8 Extensions

By treating arrays as one logical variable, the analysis is able to handle arrays as well as scalar data. This means that the analysis cannot know that some elements of an array are trusted and some are not. Either all elements are trusted or none are. This tradeoff is necessary for the abstract and constraint analyses since they are unable to compute actual offsets in the array. This tradeoff in accuracy is the same as encountered in set-based analysis [Hei93].

Records or structs can be handled by treating each field of the record as a separate variable.

Extending the language with first order procedures is simple enough. The abstract interpretation will simply model the procedure calls directly and compute fixed points in case of recursion. The constraint generation will first compute constraints for the body of a procedure and for each call add constraints matching formal and actual parameters. By copying the constraints generated for the body we can achieve a polyvariant analysis such that a particular call of the procedure with an untrustworthy argument does not influence other calls of that procedure.

A "check for trusted value" construct that will raise an error when an untrusted value is given as parameter is easily added to the language, but makes the semantics larger and a bit more complicated as it has to deal with abnormal termination. The relation between the instrumented and abstract interpretation must state that if the instrumented semantics says that a program will fail then the abstract interpretation will too. Extending the constraint generation analysis with the "check" construct means that there will only be a model for the generated constraints if all checks are met.

Extending the analysis to languages with higher order functions while still catering for pointers and mutable data seems to be more complicated and is left for future research.

The concept of trust can be extended to multiple levels of trust, so that instead of a binary lattice of trust values, a lattice with longer chains is used. For the instrumented semantics and the constraint generation, this is a straightforward generalization. For the abstract interpretation, the abstract domain is changed such that all "very trusted" pointers are below the "lesser trusted" pointers all of which are below $\top$.

2.9 Conclusion

We have argued that the analysis of the trustworthiness of data is a useful program analysis in security conscious settings, and we have given two static analyses for this purpose, one based on abstract interpretation, and another constraint-based analysis that facilitates separate analysis of program modules at the cost of slightly less accuracy.

The analyses have been proved safe with respect to an instrumented semantics that has served as the definition of the goal of the analysis.

The main contribution of this chapter is thought to be the introduction of the concept of trust analysis, and the application of it to a language with pointers and mutable data

There are some similarities between binding-time analysis [HM94] and trust analysis in this case, but there are also significant differences. Most notably, trust analysis provides the trust construct that would correspond to an unrestricted down-lift in binding-time analysis.

Acknowledgments: The author wants to thank Jens Palsberg, Peter D. Mosses and Neil D. Jones for reading earlier drafts of this chapter and giving useful comments. Also the anonymous referees provided useful feedback.

2.10 Safety of Abstract Interpretation

Fact 2.4 *If* $a \leq b$ *and* $b \in s \subseteq Val_A$ *then* $a \leq \bigvee s$.

Fact 2.5 *If* $A \vdash v \sqsubseteq a$ *and* $a \leq b$ *then* $A \vdash v \sqsubseteq b$.

Lemma 2.6 *If* $M_I \circ A \sqsubseteq A_A$ *then* $A \vdash \mathcal{E}_I\ e\ A\ M_I \sqsubseteq \mathcal{E}_A\ e\ A_A$.

Proof. By structural induction on e. We proceed with a case analysis:

- $e = i$: Show $A \vdash M_I(A(i)) \sqsubseteq A_A(i)$ which follows from the definition of $\sqsubseteq$.
- $e = [\![\mathsf{addr}\ i]\!]$: Show $A \vdash \langle A(i), \bot\rangle \sqsubseteq \{i\}$, and clearly $A(i) \in A(\{i\})$.
- $e = [\![\mathsf{deref}\ p]\!]$: Let $\langle v, t\rangle = \mathcal{E}_I\ p\ A\ M_I$ and $a = \mathcal{E}_A\ p\ A_A$, show:

  $$A \vdash \langle \pi_1(M_I(v)), t \vee \pi_2(M_I(v))\rangle \sqsubseteq \bigvee A_A(a).$$

  By induction, $A \vdash \langle v, t\rangle \sqsubseteq a$. By strong typing we can assume that v is indeed a pointer and that either $a = \top$ or $a \subseteq I$. In the first case the desired inequality holds trivially. In the second case we know that $v \in A(a)$, and also that $t = \bot$. Assume that $v = A(a_0)$, $a_0 \in a$. As $M_I \circ A \sqsubseteq A_A$, $A \vdash M_I(v) \sqsubseteq A_A(a_0)$, and using Fact 2.4 and Fact 2.5 we get the result.
- $e = [\![e_1 + e_2]\!]$: By induction,

  $$A \vdash (\langle v_1, t_1\rangle = \mathcal{E}_I\ e_1\ A\ M_I) \sqsubseteq \mathcal{E}_A\ e_1\ A_A = a_1,$$

  $$A \vdash (\langle v_2, t_2\rangle = \mathcal{E}_I\ e_2\ A\ M_I) \sqsubseteq \mathcal{E}_A\ e_2\ A_A = a_2.$$

  Show $A \vdash \langle v_1 + v_2, t_1 \vee t_2\rangle \sqsubseteq a_1 \vee a_2$. If one of $\{a_1, a_2\}$ is $\top$, the result is trivial. If they are both $\bot$, both t_1 and t_2 must be too.
- $e = [\![\mathsf{trust}\ e']\!]$: Show $A \vdash \langle \mathcal{E}_I\ e'\ A\ M_I, \bot\rangle \sqsubseteq \bot$. This follows directly from the definition of $\sqsubseteq$.
- $e = [\![\mathsf{distrust}\ e']\!]$: Trivial from the definitions.
- $e = const$: Show $A \vdash \langle const, \bot\rangle \sqsubseteq \bot$, which is trivial.

□

Lemma 2.7 *If* $M_I \circ A \sqsubseteq A_A$ *then* $A \vdash \langle addr_I\ p\ A\ M_I, \bot\rangle \sqsubseteq addr_A\ p\ A_A$

Proof. By structural induction in p.

- $p = i$: Show $A \vdash \langle A(i), \bot\rangle \sqsubseteq \{i\}$ which follows directly from the definition of $\sqsubseteq$.
- $p = [\![\mathsf{deref}\ p']\!]$: Show $A \vdash \langle \pi_1(M_I(addr_I\ p'\ A\ M_I)), \bot\rangle \sqsubseteq \bigvee A_A(addr_A\ p'\ A_A)$. By induction: $A \vdash \langle addr_I\ p'\ A\ M_I, \bot\rangle \sqsubseteq addr_A\ p'\ A_A$. If $addr_A\ p'\ A_A = \top$ the result is trivial. If $addr_A\ p'\ A_A = s \subseteq I$ then there is an identifier $a_0 \in s$ such that $A(a_0) = addr_I\ p'\ A\ M_I$. Thus $A \vdash M_I(A(a_0)) \sqsubseteq A_A(a_0)$ by the assumption that $M_I \circ A \sqsubseteq A_A$, and via Fact 2.4 and 2.5 the result follows.

□

Lemma 2.8 *$\mathcal{E}_A$ is monotone in its second argument:*

$$A_A \leq A'_A \quad \Rightarrow \quad \mathcal{E}_A\ e\ A_A \leq \mathcal{E}_A\ e\ A'_A.$$

Proof. Trivial by structural induction in e. □

Lemma 2.9 *$\mathcal{S}_A$ is monotone and increasing: If $A_A \leq A'_A$ and $t \leq t'$ then*

$$\mathcal{S}_A\ s\ A_A\ t \leq \mathcal{S}_A\ s\ A'_A\ t' \textit{ and } A_A \leq \mathcal{S}_A\ s\ A_A\ t.$$

Proof. By induction in the number of applications of $\mathcal{S}_A$, using that *asg* is monotone in all its arguments, that $addr_A$ is monotone, and that *asg* is increasing. □

Proof of Proposition 2.2 (Safety). We want to prove the following: If

$$\mathcal{S}_I\ s\ A\ M_I\ t\ =\ M',\ \ M_I \circ A \sqsubseteq A_A,\ \ \mathcal{S}_A\ s\ A_A\ t_A\ =\ A'_A \textit{ and } t \leq t_A$$

then $M' \circ A \sqsubseteq A'_A$.

The proof is by induction in the number of calls of $\mathcal{S}_I$. We proceed by a case analysis of the syntax of s:

- $s = [\![\textsf{while}\ e\ \textsf{do}\ s']\!]$: By Lemma 2.6, monotonicity of $\mathcal{E}_A$ and Fact 2.5 we know that $A \vdash \mathcal{E}_I\ e\ A\ M_I \sqsubseteq \mathcal{E}_A\ e\ A_A$.

  If v is false (in the definition of $\mathcal{S}_I$) the result follows from Lemma 2.9.

  Otherwise, let $\langle v, t' \rangle = \mathcal{E}_I\ e\ A\ M_I$, $M'' = \mathcal{S}_I\ s'\ A\ M_I\ (t \vee t')$ and $A''_A = \mathcal{S}_A\ s'\ A_A\ (t_A \vee \mathcal{E}_A\ e\ A_A)$. By the above fact on e we can apply induction and get $M'' \circ A \sqsubseteq A''_A$.

  Now we have $M' = \mathcal{S}_I\ [\![\textsf{while}\ e\ \textsf{do}\ s']\!]\ A\ M''\ t$ and

  $$A'_A = \mathcal{S}_A\ [\![\textsf{while}\ e\ \textsf{do}\ s']\!]\ A_A\ t_A = \mathcal{S}_A\ [\![\textsf{while}\ e\ \textsf{do}\ s']\!]\ A''_A\ t_A,$$

  where the last equality holds in all cases as $\mathcal{S}_A$ is increasing. By induction we get $M' \circ A \sqsubseteq A'_A$.

- $s = [\![p\ :=\ e]\!]$: Let $\langle v, t' \rangle = \mathcal{E}_I\ e\ A\ M_I$, $v_A = \mathcal{E}_A\ e\ A_A$, $a = addr_I\ p\ A\ M_I$ and $a_A = addr_A\ p\ A_A$. We need to show:

  $$M_I[\langle v, t \vee t' \rangle / a] \circ A \sqsubseteq asg\ (v_A \vee t_A)\ A_A\ a_A.$$

  If $a_A = \top$ then this follows directly from the definition of *asg*. Otherwise by Lemma 2.7 we have $A \vdash \langle a, \bot \rangle \sqsubseteq a_A$, hence there exists an $a_0 \in a_A$ such that $A(a_0) = a$. It is enough to ensure the inequality at a_0 since this is the only point where the left hand side is different from $M_I \circ A$ and *asg* is clearly monotone in the second argument so by Fact 2.5 the inequality holds automatically everywhere else. Evaluating we get:

  $$(asg\ (v_A \vee t_A)\ A_A\ a_A)(a_0) = (v_A \vee t_A \vee A_A(a_0)).$$

  and

  $$(M_I[\langle v, t \vee t' \rangle / a] \circ A)(a_0) = \langle v, t \vee t' \rangle.$$

  By Lemma 2.6 we know that $A \vdash \langle v, t' \rangle \sqsubseteq v_A$. All that remains to show is: $A \vdash \langle v, t \vee t' \rangle \sqsubseteq v_A \vee t_A \vee A_A(a_0)$ which follows from Fact 2.4.

- $s = [\![s_1; s_2]\!]$: This case follows immediately by two applications of induction.

2.11 Safety of Constraint Generation

Lemma 2.10 (Addresses) *If m is a coherent model, $m(G) = \bot$ and $A_A \sqsubseteq m$ then these two implications hold:*

$$a \in addr_A\ p\ A_A \subseteq I \quad \Rightarrow \quad m(a) = m(N(p))$$

and

$$addr_A\ p\ A_A = \top \quad \Rightarrow \quad \top = m(\eta N(p)) \leq m(N(p)).$$

Proof. By structural induction in p.

- $p = i$: $addr_A\ i\ A_A = \{i\}$ and $m(i) = m(N(p)) = m(i)$.
- $p = [\![\textsf{deref}\ p']\!]$: Note that $m(N(p')) \leq m(\eta N(p))$. First assume $a \in addr_A\ p\ A_A = \bigvee A_A(addr_A\ p'\ A_A) \subseteq I$. By induction, $b \in addr_A\ p'\ A_A \Rightarrow m(b) = m(N(p'))$. Since m is coherent, $m(\delta b) = m(\delta N(p')) = m(N(p))$. Also, as $A_A \sqsubseteq m$: $m(\delta b) = m(\Delta b) = m(a)$. Combining the equalities we get the desired result.

  Secondly, suppose $\bigvee A_A(addr_A\ p'\ A_A) = \top$. Either $addr_A\ p'\ A_A = \top$ in which case induction yields $\top = m(N(p')) \leq m(\delta N(p')) = m(N(p))$, or there is some $b_0 \in addr_A\ p'\ A_A$ such that $A_A(b_0) = \top$. Since m is a safe approximation of A_A this means $m(b_0) = \top$. By induction $m(b) = m(N(p'))$ for all $b \in addr_A\ p'\ A_A$ so we get $\top = m(b_0) = m(N(p')) \leq m(N(p))$ which is the required result.

□

Lemma 2.11 (Expressions) *The constraints generated for expressions safely approximate the abstract interpretation of expressions.*

Suppose $\langle c, v\rangle = \mathcal{E}_S\ e$, m is a coherent model of c, $A_A \sqsubseteq m$ and $a = \mathcal{E}_A\ e\ A_A$ then the following implications hold:

$$a = \top \quad \Rightarrow \quad m(v) = \top$$

and

$$a_0 \in a \subseteq I \quad \Rightarrow \quad m(a_0) = m(\delta v).$$

Proof. By structural induction in e.

- $e = i$: $\mathcal{E}_A\ i\ A_A = A_A(i)$ and $\langle c, v\rangle = \langle \emptyset, i\rangle$ by definition. If $A_A(i) = \top$ then $m(i) = m(v) = \top$ as $A_A \sqsubseteq m$. If $a_0 \in A_A(i)$ then $m(\Delta i) = m(\delta i) = m(a_0)$ by the same reason.
- $e = [\![\textsf{addr}\ i]\!]$: $\langle c, v\rangle = \langle \emptyset, \nabla i\rangle$ and $a = \{i\}$. What is required to prove thus is $m(a_0) = m(\delta v) = m(i)$ for $a_0 \in \{i\}$ which is clear.
- $e = [\![\textsf{deref}\ p]\!]$: $\langle c, v_p\rangle = \mathcal{E}_S\ p$, $v = \delta v_p$ and $a = \bigvee A_A(\mathcal{E}_A\ p\ A_A)$.

  If $a = \top$ then either $\mathcal{E}_A\ p\ A_A = \top$ and by induction $\top = m(v_p) \leq m(\delta v_p) = m(v)$, or $\mathcal{E}_A\ p\ A_A \subseteq I$ in which case there is some $a_0 \in \mathcal{E}_A\ p\ A_A$ such that $A_A(a_0) = \top$. As $A_A \sqsubseteq m$ this means that $m(a_0) = \top$. By induction $\top = m(a_0) = m(\delta v_p) = m(v)$.

  If $a_0 \in a \subseteq I$ then we must show $m(a_0) = m(\delta v)$. By induction $m(a_0') = m(\delta v_p)$ for all $a_0' \in \mathcal{E}_A\ p\ A_A \subseteq I$. $a_0 = A_A(a_0')$ for some such a_0' thus since $A_A \sqsubseteq m$, $m(\Delta a_0') = m(a_0)$ and since m is coherent:

  $$m(a_0') = m(\delta v_p) = m(v) \ \Rightarrow \ m(a_0) = m(\delta a_0') = m(\delta v).$$

- $e = [\![e_1 + e_2]\!]$: Let $\langle c_1, v_1\rangle = \mathcal{E}_S\ e_1$ and $\langle c_2, v_2\rangle = \mathcal{E}_S\ e_2$. We have $c = c_1 \cup c_2 \cup \{v_1 \leq v, v_2 \leq v\}$ and by induction the implications hold for the two subexpressions. Suppose $a = \top$: This means that $\mathcal{E}_A\ e_j\ A_A = \top$ for some $j \in \{1, 2\}$ and by induction this means that $m(v_j) = \top$ and by definition of c we get $m(v) = \top$.

  By strong typing, the abstract value for the expression must be either $\top$ or $\bot$ so this concludes the case.

- $e = [\![\mathsf{trust}\ e']\!]$: We have $a = \bot$ so the implications hold vacuously.

- $e = [\![\mathsf{distrust}\ e']\!]$: We have $a = \top$ and $c = \{\top \leq v\}$ hence $m(v) = \top$ as required.

- $e = \mathit{const}$: We have $a = \bot$ so the implications hold vacuously.

□

Proof of Proposition 2.3. We want to prove the following: If

$$\begin{aligned} \langle c, v\rangle &= \mathcal{S}_S\ s, && (2.1)\\ m &\models c, \textit{ and } m \textit{ is coherent} && (2.2)\\ A_A &\sqsubseteq m, && (2.3)\\ \forall x \in v &: t \leq m(x), && (2.4)\\ A'_A &= \mathcal{S}_A\ s\ A_A\ t && (2.5) \end{aligned}$$

then $A'_A \sqsubseteq m$.

Proof. We proceed by induction in the number of calls to $\mathcal{S}_A$. If $m(G) = \top$ then the final inequality holds regardless of A'_A, so assume $m(G) = \bot$. A case analysis follows:

- $s = [\![\mathsf{while}\ e\ \mathsf{do}\ s']\!]$: Let $\langle c_e, v_e\rangle = \mathcal{E}_S\ e$ and $\langle c_s, v_s\rangle = \mathcal{S}_S\ s'$. By definition of c: $x \in v_s \Rightarrow m(v_e) \leq m(x)$ and by Lemma 2.11 $\mathcal{E}_A\ e\ A_A = \top \Rightarrow m(v_e) = \top$ thus by (2.4) $\forall x \in v_s : t \vee \mathcal{E}_A\ e\ A_A \leq m(x)$. We can now apply induction on s' and get $A'_A = \mathcal{S}_A\ s'\ A_A\ (t \vee \mathcal{E}_A\ e\ A_A) \sqsubseteq m$. If this is the same as A_A we are done. Otherwise we apply induction once more and get the result.

- $s = [\![p\ := \ e]\!]$: If $\mathit{addr}_A\ p\ A_A = \top$ then by Lemma 2.10, $\top = m(\eta N(p)) \leq m(G)$ so in that case $A'_A \sqsubseteq m$ by definition of $\sqsubseteq$.

  Now suppose $a_0 \in a = \mathit{addr}_A\ p\ A_A \subseteq I$. A'_A differs from A_A only on the set a by definition of asg. Let $\langle c_e, v_e\rangle = \mathcal{E}_S\ e$ and $a_e = \mathcal{E}_A\ e\ A_A$.

  If $A'_A(a_0) = t \vee A_A(a_0) \vee a_e = \top$ we must show $m(a_0) = \top$. By (2.4) $t \leq m(N(p)) = m(a_0)$ where that last equality comes from Lemma 2.10. By (2.3) $A_A(a_0) = \top \Rightarrow m(a_0) = \top$. By Lemma 2.11 $a_e = \top \Rightarrow m(v_e) = \top$, and by definition of c, $m(v_e) \leq m(v) = m(N(p)) = m(a_0)$, using Lemma 2.10 last. For $A'_A(a_0)$ to be $\top$ at least one of the parts of the above disjunction must be $\top$ (by definition of the Val_A lattice) and by the inequalities, $m(a_0) = \top$ in all cases.

  If $A'_A(a_0) = t \vee A_A(a_0) \vee a_e \subseteq I$ then we must show that $a' \in A'_A(a_0) \Rightarrow m(\Delta a_0) = m(a')$. a' cannot belong to t as $t \in \mathit{Tr}$. If $a' \in A_A(a_0)$ then (2.3) secures the result. Otherwise, if $a' \in a_e$ then by Lemma 2.11 $m(a') = m(\delta v_e) = m(\delta N(p))$ where the last equality stems from the definition of c. By Lemma 2.10 and coherence $m(\delta N(p)) = m(\delta a_0) = m(\Delta a_0)$.

- $s = [\![s_1; s_2]\!]$: Let $A''_A = \mathcal{S}_A\ s_1\ A_A\ t$ and $\langle c_1, v_1\rangle = \mathcal{S}_S\ s_1$. Now $c_1 \subseteq c$ and $v_1 \subseteq v$ by definition of $\mathcal{S}_S$, so by induction we get $A''_A \sqsubseteq m$. With this and equivalent considerations as above we can apply induction to A''_A and s_2 and get $A'_A \sqsubseteq m$ as required.

□

Chapter 3

Trust in the λ-calculus

You may be deceived if you trust too much,
but you will live in torment if you don't trust enough.
Frank Crane

This chapter is joint work with Jens Palsberg.

3.1 Introduction

In the previous chapter the concept of trust analysis was introduced for a first order imperative language using abstract interpretation and constraints. This chapter investigates trust analysis as a type system for a pure functional language based on the λ-calculus.

The remainder of the chapter is structured as follows. First we give some intuitions about the intended program analysis, the semantics of our example language, and the type system. Then we present an extension of the λ-calculus together with an operational reduction calculus. The calculus is shown to have the Church-Rosser property. We also give a denotational semantics for our language and relate it to the reduction rules. We define our static trust analysis in terms of a type system. The type system is shown to have the Subject Reduction property with respect to the reduction rules of the semantics. We then relate our type system to the classical Curry type system for λ-calculus and obtain two simulation theorems relating reductions in our calculus to reductions in classical λ-calculus, and finally we prove that well-typed terms are strongly normalizing. Then a type inference algorithm is presented and proved correct with respect to the type system. Finally we discuss how to extend the type system to handle recursion, modules and polymorphism, and we relate trust analysis to other program analyses.

3.1.1 Intuitions and Motivation

The method we propose to help the programmer ensure that he inserts checks on all the required data-paths in his program is here formulated in terms of an annotated type-system. This type-system must be able to type both trusted data and untrusted data as both kinds of data occur naturally in most programs, but still it must be able to distinguish between these two kinds of data.

As this chapter is concerned with a higher-order language, functions are data as well so a function is itself either trusted or untrusted. Intuitively it should be clear that if we apply an untrustworthy function then the result of that application is untrustworthy as well, since the function itself may depend on untrustworthy information from the outside. This leads us to our type rule for function application as shown in Figure 3.7.

Our type system starts from the simply typed λ-calculus and annotates each type constructor occurring in the program with a trustworthiness. A trustworthy boolean has the annotated type $\mathsf{Bool}^{\mathsf{tr}}$, and an untrustworthy function sending trusted booleans to trusted booleans has the annotated type $(\mathsf{Bool}^{\mathsf{tr}} \rightarrow \mathsf{Bool}^{\mathsf{tr}})^{\mathsf{dis}}$. Allowing untrustworthy functions to return trusted values in this sense, allows us to avoid complicated well-formedness constraints on types. Instead we let our application rule handle the problem.

Suppose f is a function that can accept an untrusted argument. This must mean that f cannot use that argument in places where a trusted value is required unless it does some checking beforehand on the argument. If we give f a trusted argument then these checks should succeed which means that if a function can accept untrusted input then it can also accept trusted input. This leads us to a type system with subtyping, such that an expression of a trusted type can be typed as an untrusted type. The ordering between base-types as determined by their trustworthiness is extended to higher types using the usual contra/co-variant structural subtyping idea of Mitchell [Mit84], Fuh and Mishra [FM90], Cardelli [Car84] and others.

The trust type system differs from many other type systems in that given a bare value (eg. 7) it is not possible to see by just examining the value whether it is trustworthy or not. This is where our three extensions to the basic λ-calculus come in. They basically link the dynamic semantics of the calculus to our type system, and they can be seen as a kind of annotations to the program that the type inference algorithm will attempt to verify for consistency. The trust E construct indicates to the type system that the result of E may now be trusted. Dually, distrust E indicates that the result of E cannot be trusted, for example after a failing validity check, and last but not least: the check E construct indicates to the analysis that the result of E is *required* to be trustworthy. Well-written programs will only have a few syntactic places where the three new constructs are used. The type system propagates the trust information though the program, ensuring that for any instance of check E in a well-typed program the expression E is statically known to be trustworthy.

3.1.2 An Example

Consider the following piece of code written in an SML like syntax for a network server program:

```
read_from_network :: (Client^dis → (Req^dis, Sig^tr))^tr
verify_signature :: (Sig^tr → Bool^tr)^tr
handle_event :: (Req^tr → Unit^tr)^tr
handle_wrong_signature :: ((Req^dis, Sig^tr)→ Unit^tr)^tr

fun get_request client =
    let (req, signature) = read_from_network(client) in
        if verify_signature(signature) then
          handle_event(trust req)
```

$$E, F, G, H \quad ::= \quad x \mid \lambda x.E \mid EE \mid \mathsf{trust}\ E \mid \mathsf{distrust}\ E \mid \mathsf{check}\ E$$

Figure 3.1: The syntax of expressions

```
else
  handle_wrong_signature(req, signature).
```

The server first reads a packet from the network client and if the signature of the packet can be verified, then we can trust the request data and the event handler is called with the request part of the packet. In case the signature cannot be verified an error handler is called which may eventually display some error message on the client's display.

The event handler could be called from many places in the program and to avoid by accident calling it with a request that has not yet been verified, we use the trust type system to require that the handler gets only trustworthy requests. The handler code may then look something like:

```
fun handle_event req =
    let trusted_req = check req
    in ...
```

and it will therefore get the argument type $\mathsf{Req}^{\mathsf{tr}}$. Notice the small number of program annotations of the form trust E and check E as opposed to the numerous trust annotations on the types that are inferred by our inference algorithm.

3.2 Syntax and Semantics

This section presents the syntax and operational semantics of the trust language. We have chosen to formalize our analysis in an extension of the traditional λ-calculus without any predetermined reduction order, that is, we study a pure calculus to gain results that will specialize equally well to call-by-value implementations as to call-by-need implementations. The drawback of this is of course that some results require more work, for example we need a lengthy proof of the Church-Rosser property of our calculus, something that would come essentially for free had we chosen to study just one specific reduction strategy such as left-most inner-most reduction for call-by-value. Figure 3.1 defines the syntax of our language.

Variables, λ-abstraction and application behave as usual. The trust E construct is used to introduce trusted values in a program. Symmetrically, distrust indicates untrusted values. The check construct will reduce only on trusted values, so evaluation may get *stuck* if an expression check(distrust E) occurs at some point during evaluation. According to the previous section, the three new constructs should be regarded as the interface between the dynamic semantics of the program and the static analysis (the type system). But in order to prove the soundness of the analysis we have to give some dynamic meaning to the new constructs. This is done in terms of fairly obvious reduction rules for the constructs defining how they interact. One can also see the new constructs as operating on tags associated with all values at run-time, and this is the view taken in the denotational semantics given later.

3.2.1 Reduction Rules

The reduction (or evaluation) rules for the language are given in Figure 3.2. Stating $E \to E'$ means that there is a derivation of that reduction in the system.

There are three kinds of values around during reduction: trusted, distrusted and untagged. Untagged lambdas are treated as trusted program constants in the (Lambda Contraction) rules, since lambdas stem from the program text which the programmer is writing himself and they may therefore be trusted. As discussed in Section 3.5 this may change in a larger scale situation with modules.

In order to facilitate the proof of the Church-Rosser property of the system, the reduction rules form a reflexive "one step" transition relation. This is inspired by the proof of Church-Rosser for the ordinary λ-calculus by Tait and Martin-Löf in [Bar81, pp. 59–62].

The contraction rules exist to eliminate redundant uses of our new constructs in the calculus. For example, trusting an expression twice is the same as trusting it once (the first (Trust Contraction) rule) and checking the trustworthiness of an expression then explicitly trusting it is the same as just checking the expression (Check Contraction). Checking an explicitly trusted expression succeeds and yields a trusted expression. Note that there is no rule contracting distrust(check E) since this would allow the removal of the check on the trustworthiness of E. The reason this particular choice of reduction rules is that we want check to act in a "call-by-value" fashion as discussed in the next section.

In the following we always consider equality of terms modulo α-renaming. Since we are working with a reflexive reduction relation we have to be careful in our definition of what is meant by a normal form. A *significant reduction* $E \to F$ is a reduction whose derivation uses at least one of the contraction rules or a β-rule. A term E is said to be in *normal form* when there are no significant reductions[1] starting from E. There are *proper* and *improper* normal forms. A normal form containing a sub-term of the form check(distrust E) is said to be improper. All other normal forms are proper. We will write $\to^*$ for the reflexive transitive closure of the reduction relation $\to$.

As an example of how to extend the language with usual programming constructs, we show in Figure 3.3 how a reduction rule for program constants would look and the derived rules we get for if-then-else with the usual coding of booleans in the λ-calculus. Notice how the (Constant) rules are patterned after the (Lambda Contraction) rules, and how the trustworthiness of the condition in an if-then-else construct affects the trustworthiness of the result. It shows that our function application rule seamlessly handles what Denning in [Den76] called indirect data dependences (implicit flow). In Section 3.5 we also show how to encode a rec construct in the language.

3.2.2 The Nature of check

The contraction rules that we have in the case where check is the inner construction are given by the (Check Contraction) rules:

$$\frac{E \to \mathsf{check}\ F}{\begin{array}{c}\mathsf{trust}\ E \to \mathsf{check}\ F \\ \mathsf{check}\ E \to \mathsf{check}\ F\end{array}}$$

[1] A reduction $E \to F$ may be significant even when $E = F$: $\Omega = (\lambda x.xx)(\lambda x.xx) \to \Omega$ is a significant reduction.

$$E \to E \qquad \text{(Reflex)}$$

$$\frac{E \to E'}{\begin{array}{ccc} \lambda x.E & \to & \lambda x.E' \\ \mathsf{trust}\ E & \to & \mathsf{trust}\ E' \\ \mathsf{distrust}\ E & \to & \mathsf{distrust}\ E' \\ \mathsf{check}\ E & \to & \mathsf{check}\ E' \end{array}} \qquad \text{(Sub)}$$

$$\frac{E \to \mathsf{trust}\ E'}{\begin{array}{ccc} \mathsf{trust}\ E & \to & \mathsf{trust}\ E' \\ \mathsf{distrust}\ E & \to & \mathsf{distrust}\ E' \\ \mathsf{check}\ E & \to & \mathsf{trust}\ E' \end{array}} \qquad \text{(Trust Contraction)}$$

$$\frac{E \to \mathsf{distrust}\ E'}{\begin{array}{ccc} \mathsf{trust}\ E & \to & \mathsf{trust}\ E' \\ \mathsf{distrust}\ E & \to & \mathsf{distrust}\ E' \end{array}} \qquad \text{(Distrust Contraction)}$$

$$\frac{E \to \mathsf{check}\ E'}{\begin{array}{ccc} \mathsf{trust}\ E & \to & \mathsf{check}\ E' \\ \mathsf{check}\ E & \to & \mathsf{check}\ E' \end{array}} \qquad \text{(Check Contraction)}$$

$$\frac{E \to \lambda x.E'}{\begin{array}{ccc} \mathsf{trust}\ E & \to & \lambda x.E' \\ \mathsf{check}\ E & \to & \lambda x.E' \end{array}} \qquad \text{(Lambda Contraction)}$$

$$\frac{E \to E' \qquad F \to F'}{\begin{array}{ccc} EF & \to & E'F' \\ (\lambda x.E)F & \to & E'[F'/x] \\ (\mathsf{distrust}\ (\lambda x.E))F & \to & \mathsf{distrust}\ E'[F'/x] \end{array}} \qquad \text{(Application)}$$

Figure 3.2: The reduction rules.

$$\frac{E \to const}{\begin{array}{c}\mathsf{trust}\ E \to const\\ \mathsf{check}\ E \to const\end{array}} \quad \text{(Constant)}$$

$$\begin{array}{rl}\mathsf{T} \equiv \mathsf{K} \equiv \lambda xy.x & \text{(True)}\\ \mathsf{F} \equiv \lambda xy.y & \text{(False)}\\ \mathsf{if}\ E\ \mathsf{then}\ F\ \mathsf{else}\ G \equiv EFG & \text{(If)}\end{array}$$

$$\frac{E \to E' \quad F \to F'}{\begin{array}{c}\mathsf{if}\ \mathsf{T}\ \mathsf{then}\ E\ \mathsf{else}\ F \to^* E'\\ \mathsf{if}\ \mathsf{F}\ \mathsf{then}\ E\ \mathsf{else}\ F \to^* F'\\ \mathsf{if}\ \mathsf{distrust}\ \mathsf{T}\ \mathsf{then}\ E\ \mathsf{else}\ F \to^* \mathsf{distrust}\ E'\\ \mathsf{if}\ \mathsf{distrust}\ \mathsf{F}\ \mathsf{then}\ E\ \mathsf{else}\ F \to^* \mathsf{distrust}\ F'\end{array}} \quad \text{(If)}$$

Figure 3.3: Example rules.

and most notably, there is no rule for contracting distrust(check E). There is at least one other set of rules for this case that may come to mind, namely this set of rules:

$$\frac{E \to \mathsf{check}\ F}{\begin{array}{c}\mathsf{trust}\ E \to \mathsf{trust}\ F\\ \mathsf{distrust}\ E \to \mathsf{distrust}\ F\\ \mathsf{check}\ E \to \mathsf{check}\ F\end{array}}$$

The intuition for the first of these alternative rules is that if we put trust around some expression there is really no need to perform the check inside the trust construct since the program is going to trust the resulting value anyway. The second rule is the symmetric case, and is needed to make the resulting calculus Church-Rosser. The last rule also occurs in our system. The calculus that results from this alternative set of rules makes fewer programs end up in a stuck configuration, because it is now possible to place a stuck expression in a context that will make it reducible, as in trust(check(distrust E)) which is stuck in our calculus, but reduces to trust E under the alternative rules.

One way to think about this is to view our definition of check as "call-by-value" in that it really needs to see the (possibly implicit) tag on its subexpression before it can be reduced away, whereas with the alternative rules, check is "call-by-name" in that it can be reduced away, depending on its context, without considering the trustworthiness of its argument. The "call-by-value" nature of check is also reflected in the denotational semantics of check given later. Note, however, that for the core λ-calculus we have the full β-rule. It's only the new construct, check, that behaves in a "call-by-value" or "call-by-name" fashion.

In our view the alternative rules are less intuitive to the programmer, in that if he writes check somewhere in the program he probably wants it to check the trustworthiness of its argument regardless of the surrounding context. But one can argue both ways: the "call-by-name" version of check might be the most natural choice in a lazy implementation of a functional language such as Haskell, whereas the "call-by-value" version might be most suitable for a call-by-value language such as SML. Either way it's easy to alter the proof of

the Subject Reduction theorem (Theorem 3.11) for our type system to the alternative rules, so our type system is sound for both sets of rules.

3.2.3 Church-Rosser

The Church-Rosser (confluence) theorem for a reduction system states that for any term, if the term can reduce to two different terms there exists a successor term such that both of the two reduced terms can further reduce to that common successor. A corollary of this is that a normal form is unique if it exists.

Theorem 3.1 (Church-Rosser) *For expressions E, F and G. If $E \rightarrow^* F$ and $E \rightarrow^* G$ then there is an expression H such that $F \rightarrow^* H$ and $G \rightarrow^* H$.*

Proof. By the Diamond lemma below (Lemma 3.7) and Lemma 3.2.2 of [Bar81]. □

Lemma 3.2 *If $E \rightarrow F$ and E has a certain structure then some conditions on the structure of F hold, as made explicit below.*

- *If $E = \mathsf{trust}\ E_1 \rightarrow F$ then $F = \alpha\ F_1$ where $\alpha \in \{\mathsf{check}, \mathsf{trust}, \lambda x.\}$.*
- *If $E = \mathsf{check}\ E_1 \rightarrow F$ then $F = \alpha\ F_1$ where $\alpha \in \{\mathsf{check}, \mathsf{trust}, \lambda x.\}$.*
- *If $E = \mathsf{distrust}\ E_1 \rightarrow F$ then F is of the form $\mathsf{distrust}\ F_1$.*
- *If $E = \lambda x.E_1 \rightarrow F$ then F is of the form $\lambda x.F_1$.*

Proof. In each case by inspection of the reduction rules. □

Lemma 3.3 (Trust/Check Identity) *Let $\alpha \in \{\mathsf{trust}, \mathsf{check}, \lambda x.\}$.*
If $E \rightarrow \alpha\ E_1$ then $\mathsf{check}\ E \rightarrow \alpha\ E_1$ and $\mathsf{trust}\ E \rightarrow \alpha\ E_1$

Proof. By inspection of the reduction rules, especially the contraction rules. □

Lemma 3.4 (Symmetry) *Let $\alpha, \beta \in \{\mathsf{trust}, \mathsf{distrust}\}$. If $\alpha\ E \rightarrow \alpha\ E'$ then $\beta\ E \rightarrow \beta\ E'$*

Proof. By induction on the structure of the derivation of $\alpha\ E \rightarrow \alpha\ E'$, verifying that in each case there is also a corresponding rule for the opposite combination. □

Lemma 3.5 (Pre-Substitution) *If $E \rightarrow F$ then $G[E/x] \rightarrow G[F/x]$.*

Proof. By induction on the structure of G. This is essentially a consequence of the (Sub) rules and the first (Application) rule. □

Lemma 3.6 (Substitution) *If $E \rightarrow F$ and $G \rightarrow H$ then $E[G/x] \rightarrow F[H/x]$.*

Proof. By induction on the structure of the derivation of $E \rightarrow F$. If $F = E$ by the (Reflex) axiom, we must show that $E[G/x] \rightarrow E[H/x]$ given that $G \rightarrow H$. This is the Pre-Substitution lemma (3.5).

For all the rules except the (Application) case: Suppose that α, β, and γ are in the set $\{\mathsf{check}, \mathsf{trust}, \mathsf{distrust}, \lambda x.\}$ as appropriate, and β and γ may be empty as well. Assume the rule

$$\frac{E_1 \to \beta\ F_1}{E = \alpha\ E_1 \to \gamma\ F_1 = F}$$

is the last rule in the derivation of $E \to F$. By the induction hypothesis we get

$$E_1[G/x] \to (\beta\ F_1)[H/x] = \beta(F_1[H/x]).$$

Now $E[G/x] = \alpha\ E_1[G/x]$ and $F[H/x] = \gamma\ F_1[H/x]$, and we may now deduce

$$\frac{E_1[G/x] \to \beta\ F_1[H/x]}{\alpha\ E_1[G/x] \to \gamma\ F_1[H/x]}$$

as required. Of course, in the case of a lambda, if the bound variable is the one substituted for, nothing happens during substitution, i.e.

$$E[G/x] = E \to F = F[H/x]$$

as the only rule applicable to the case of $\alpha = \lambda x.$ is the (Sub) rule.

The (Application) cases: If $E = E_1E_2 \to F_1F_2 = F$, where $E_1 \to F_1$ and $E_2 \to F_2$ then the result follows directly from the induction hypothesis. Suppose the last rule in the derivation of $E \to F$ was $(x \neq y)$:

$$\frac{E_1 \to F_1 \quad E_2 \to F_2}{E = (\lambda y.E_1)E_2 \to F_1[F_2/y] = F}$$

By the induction hypothesis $E_1[G/x] \to F_1[H/x]$ and similarly for E_2. Also $E[G/x] = (\lambda y.E_1[G/x])(E_2[G/x])$ and

$$F[H/x] = (F_1[F_2/y])[H/x] = (F_1[H/x])[F_2[H/x]/y]$$

where the last equality depends on y not being free in H. This can be assured by α-renaming H. We may now deduce:

$$\frac{E_1[G/x] \to F_1[H/x] \quad E_2[G/x] \to F_2[H/x]}{(\lambda y.E_1[G/x])(E_2[G/x]) \to (F_1[H/x])[F_2[H/x]/y]}$$

Suppose the last rule in the derivation of $E \to F$ was:

$$\frac{E_1 \to F_1 \quad E_2 \to F_2}{E = (\lambda x.E_1)E_2 \to F_1[F_2/x] = F}$$

By the induction hypothesis, $E_2[G/x] \to F_2[H/x]$. Also $E[G/x] = (\lambda x.E_1)(E_2[G/x])$ and

$$F[H/x] = (F_1[F_2/x])[H/x] = F_1[F_2[H/x]/x].$$

Now

$$\frac{E_1 \to F_1 \quad E_2[G/x] \to F_2[H/x]}{(\lambda x.E_1)(E_2[G/x]) \to F_1[F_2[H/x]/x]}$$

as required. Two similar cases apply to the $\mathsf{distrust}\ \lambda x.E$ case. □

Lemma 3.7 (Diamond) *For expressions E, F and G. If $E \to F$ and $E \to G$ then there is an expression H such that $F \to H$ and $G \to H$.*

Proof. By induction on the derivation of $E \to F$ and $E \to G$ and by cases on how F and G must look depending on E.

Depending on E there are a number of applicable rules. In all cases (Reflex) and (Sub) are applicable.

1. $E = \lambda x.E_1$: none other.

2. $E = \mathsf{trust}\ E_1$:

   (a) $E \to \mathsf{trust}\ E_1'$ when $E_1 \to \mathsf{trust}\ E_1'$. (Trust Contraction)

   (b) $E \to \lambda x.E_1'$ when $E_1 \to \lambda x.E_1'$. (Lambda Contraction)

   (c) $E \to \mathsf{check}\ E_1'$ when $E_1 \to \mathsf{check}\ E_1'$. (Check Contraction)

   (d) $E \to \mathsf{trust}\ E_1'$ when $E_1 \to \mathsf{distrust}\ E_1'$. (Distrust Contraction)

3. $E = \mathsf{distrust}\ E_1$:

   (a) $E \to \mathsf{distrust}\ E_1'$ when $E_1 \to \mathsf{trust}\ E_1'$. (Trust Contraction)

   (b) $E \to \mathsf{distrust}\ E_1'$ when $E_1 \to \mathsf{distrust}\ E_1'$. (Distrust Contraction)

4. $E = \mathsf{check}\ E_1$:

   (a) $E \to \mathsf{trust}\ E_1'$ when $E_1 \to \mathsf{trust}\ E_1'$. (Trust Contraction)

   (b) $E \to \mathsf{check}\ E_1'$ when $E_1 \to \mathsf{check}\ E_1'$. (Check Contraction)

   (c) $E \to \lambda x.E_1'$ when $E_1 \to \lambda x.E_1'$. (Lambda Contraction)

5. $E = (\lambda x.E_1)E_2$: $E \to E_1'[E_2'/x]$ when $E_1 \to E_1'$ and $E_2 \to E_2'$.

6. $E = (\mathsf{distrust}\ \lambda x.E_1)E_2$: $E \to \mathsf{distrust}\ E_1'[E_2'/x]$ when $E_1 \to E_1'$ and $E_2 \to E_2'$.

If $E \to F$ or $E \to G$ by (Reflex) then there is no problem, one may just use the rule applied in the other branch to get to the common successor. Case 1 is easy as well: there is only one applicable rule except (Reflex) namely (Sub).

The tables below map pairs of "outgoing" reductions to proofs of the corresponding case.

Case 2	2a	2b	2c	2d	(Sub)
2a		B	B	C	A
2b			B	C	A
2c				C	A
2d					D

Case 3	3a	3b	(Sub)
3a		G	F
3b			E

Case 4	4a	4b	4c	(Sub)
4a		B	B	A
4b			B	A
4c				A

For case 6 the argument is as follows: Here the last rules in the derivation of $E \to F$ and $E \to G$ were:

$$\frac{E_1 \to F_1 \quad E_2 \to F_2}{E = (\mathsf{distrust}\ \lambda x.E_1)E_2 \to \mathsf{distrust}\ F_1[F_2/x] = F}(\text{Application})$$

and

$$\frac{E_1 \to G_1 \quad E_2 \to G_2}{E = (\mathsf{distrust}\ \lambda x.E_1)E_2 \to (\mathsf{distrust}\ \lambda x.G_1)G_2 = G}(\text{Sub})$$

respectively. By the induction hypothesis there are H_1 and H_2 such that $F_1 \to H_1$, $G_1 \to H_1$ and $F_2 \to H_2$, $G_2 \to H_2$. So by the Substitution lemma (Lemma 3.6) (and (Sub)):

$$F = \mathsf{distrust}\ F_1[F_2/x] \to \mathsf{distrust}\ H_1[H_2/x] = H$$

and by (Application)

$$G = (\mathsf{distrust}\ \lambda x.G_1)G_2 \to \mathsf{distrust}\ H_1[H_2/x] = H.$$

Case 5 without distrust is similar.

In each of the cases below, the quest is to find an appropriate common successor H to F and G.

Case A. Let $\alpha \in \{\mathsf{trust}, \mathsf{check}\}$ and $\beta \in \{\mathsf{trust}, \mathsf{check}, \lambda x.\}$. The last rules of the derivation of $E \to F$ and $E \to G$ were

$$\frac{E_1 \to \beta\ F_1}{E = \alpha\ E_1 \to \beta\ F_1 = F}(\beta\ \text{Contraction})$$

$$\frac{E_1 \to G_1}{E = \alpha\ E_1 \to \alpha\ G_1 = G}(\text{Sub})$$

By the induction hypothesis there is an H_1 such that $G_1 \to H_1$ and $\beta\ F_1 \to H_1$. By Lemma 3.2 and the restriction on β; $H_1 = \gamma\ H_2$ where $\gamma \in \{\mathsf{trust}, \mathsf{check}, \lambda x.\}$. By the Trust/Check Identity lemma (Lemma 3.3), $G_1 \to \gamma\ H_2$ implies that $\alpha\ G_1 \to \gamma\ H_2 = H_1$. So we can use $H = H_1$.

Case B. Let $\alpha, \beta, \gamma \in \{\mathsf{trust}, \mathsf{check}, \lambda x\}$ as appropriate. The last rules used in the derivation of $E \to F$ and $E \to G$ are:

$$\frac{E_1 \to \beta\ F_1}{E = \alpha\ E_1 \to \beta\ F_1 = F}(\beta\ \text{Contraction})$$

$$\frac{E_1 \to \gamma\ G_1}{E = \alpha\ E_1 \to \gamma\ G_1 = G}(\gamma\ \text{Contraction})$$

By the induction hypothesis there is an H_1 such that $\beta\ F_1 \to H_1$ and $\gamma\ G_1 \to H_1$. We may now use H_1 as H.

Case C. Let $\alpha \in \{\mathsf{trust}, \mathsf{check}, \lambda x.\}$. The two last rules used in the derivation of $E \to F$ and $E \to G$ are:

$$\frac{E_1 \to \alpha\ F_1}{E = \mathsf{trust}\ E_1 \to \alpha\ F_1 = F}(\alpha \text{ Contraction})$$

$$\frac{E_1 \to \mathsf{distrust}\ G_1}{E = \mathsf{trust}\ E_1 \to \mathsf{trust}\ G_1 = G}(\text{Distrust Contraction})$$

By the induction hypothesis we know there exists H_1 such that $\alpha\ F_1 \to H_1$. Here Lemma 3.2 says that $H_1 = \beta\ H_2$ where $\beta \in \{\mathsf{check}, \mathsf{trust}, \lambda x.\}$. Also by the induction hypothesis we have $\mathsf{distrust}\ G_1 \to H_1$. And here Lemma 3.2 says that $H_1 = \mathsf{distrust}\ H_2$! This is a contradiction so it cannot be the case that both $E_1 \to \alpha\ F_1$ and $E_1 \to \mathsf{distrust}\ G_1$.

Case D. The two last rules used in the derivation of $E \to F$ and $E \to G$ are:

$$\frac{E_1 \to \mathsf{distrust}\ F_1}{E = \mathsf{trust}\ E_1 \to \mathsf{trust}\ F_1 = F}(\text{Distrust Contraction})$$

$$\frac{E_1 \to G_1}{E = \mathsf{trust}\ E_1 \to \mathsf{trust}\ G_1 = G}(\text{Sub})$$

By the induction hypothesis there is an H_1 such that $\mathsf{distrust}\ F_1 \to H_1$ and $G_1 \to H_1$. By Lemma 3.2 $H_1 = \mathsf{distrust}\ H_2$ so

$$\frac{G_1 \to \mathsf{distrust}\ H_2}{\mathsf{trust}\ G_1 \to \mathsf{trust}\ H_2}(\text{Distrust Contraction})$$

By Symmetry (Lemma 3.4) $\mathsf{distrust}\ F_1 \to \mathsf{distrust}\ H_2$ implies $\mathsf{trust}\ F_1 \to \mathsf{trust}\ H_2$. So we can use $\mathsf{trust}\ H_2$ as H.

Case E. The two last rules used in the derivation of $E \to F$ and $E \to G$ are:

$$\frac{E_1 \to \mathsf{distrust}\ F_1}{E = \mathsf{distrust}\ E_1 \to \mathsf{distrust}\ F_1 = F}(\text{Distrust Contraction})$$

$$\frac{E_1 \to G_1}{E = \mathsf{distrust}\ E_1 \to \mathsf{distrust}\ G_1 = G}(\text{Sub})$$

By the induction hypothesis there is an H_1 such that $\mathsf{distrust}\ F_1 \to H_1$ and $G_1 \to H_1$. By Lemma 3.2 $H_1 = \mathsf{distrust}\ H_2$. By (Distrust Contraction) $G_1 \to \mathsf{distrust}\ H_2$ implies $\mathsf{distrust}\ G_1 \to \mathsf{distrust}\ H_2 = H_1$. So we use $H = H_1$ in this case.

Case F. The two last rules used in the derivation of $E \to F$ and $E \to G$ are:

$$\frac{E_1 \to \mathsf{trust}\ F_1}{E = \mathsf{distrust}\ F_1 \to \mathsf{distrust}\ F_1 = F}(\text{Trust Contraction})$$

$$\frac{E_1 \to G_1}{E = \mathsf{distrust}\ E_1 \to \mathsf{distrust}\ G_1 = G}(\text{Sub})$$

$$
\begin{array}{rcl}
D & = & (((D \to D) \oplus \mathsf{base}) \otimes \{\mathsf{tr}, \mathsf{dis}\}_\perp) \oplus \{\mathsf{error}\}_\perp. \\
\mathit{Env} & = & \mathit{Var} \to D. \\
[\![\cdot]\!] & \in & E \to \mathit{Env} \to D \\
E, F & \in & E \\
x & \in & \mathit{Var} \\
\rho & \in & \mathit{Env}.
\end{array}
$$

Figure 3.4: Domain equations.

By the induction hypothesis there is an H_1 such that $\mathsf{trust}\ F_1 \to H_1$ and $G_1 \to H_1$. By Lemma 3.2 $H_1 = \alpha\ H_2$ where $\alpha \in \{\mathsf{trust}, \mathsf{check}, \lambda x.\}$.

If $\alpha = \mathsf{trust}$ then we have $G_1 \to \mathsf{trust}\ H_2$ and $\mathsf{trust}\ F_1 \to \mathsf{trust}\ H_2$ and by Symmetry $\mathsf{distrust}\ F_1 \to \mathsf{distrust}\ H_2$. By (Trust Contraction) we also get $\mathsf{distrust}\ G_1 \to \mathsf{distrust}\ H_2$, so here we may use $H = \mathsf{distrust}\ H_2$.

If $\alpha = \mathsf{check}$ or $\alpha = \lambda x.$ then by (Sub) we get $\mathsf{distrust}\ G_1 \to \mathsf{distrust}\ (\alpha\ H_2)$. Since $\mathsf{trust}\ F_1 \to \alpha\ H_2$ one sees by inspection of the rules that for each α there is just one possible last rule for this reduction so we must have $F_1 \to \alpha\ H_2$. Now by (Sub) we get $\mathsf{distrust}\ F_1 \to \mathsf{distrust}\ (\alpha\ H_2)$. So here we may use $H = \mathsf{distrust}\ (\alpha\ H_2)$.

Case G. The two last rules used in the derivation of $E \to F$ and $E \to G$ are:

$$
\frac{E_1 \to \mathsf{distrust}\ F_1}{E = \mathsf{distrust}\ E_1 \to \mathsf{distrust}\ F_1 = F}\text{(Distrust Contraction)}
$$

$$
\frac{E_1 \to \mathsf{trust}\ G_1}{E = \mathsf{distrust}\ E_1 \to \mathsf{distrust}\ G_1 = G}\text{(Trust Contraction)}
$$

By the induction hypothesis there is an H_1 such that $\mathsf{distrust}\ F_1 \to H_1$ and $\mathsf{trust}\ G_1 \to H_1$, but by Lemma 3.2 this is a contradiction so this case cannot arise.

This concludes the proof of the Diamond lemma. □

3.2.4 Denotational Semantics

In this section we give a denotational semantics for the calculus and prove it sound with respect to the calculus. The denotational semantics has a purely expositional purpose. Some readers may find the direct style denotational semantics easier to comprehend than the reduction rules. However, as we saw in the section on the nature of check, the reduction rules are easy to alter to get a different, but justifiable, semantics of check, whereas the denotational semantics for the alternative set of rules would be substantially different, as we would have to abandon the direct style and instead use a continuation based semantics.

In Figure 3.4 we define our semantic domains, using coalesced sums and smash products. A value is either error (standing for a semantic error) or a pair consisting of the “real” value (a function or a value of base-type from the base domain) and a trust tag, either tr or dis. The domain $\{\mathsf{tr}, \mathsf{dis}\}_\perp$ is the usual flat three-point domain where $\perp$ is the bottom element according to the $\sqsubseteq$ ordering. For the definition of application we define *another* ordering

$$
\begin{array}{lcl}
[\![x]\!]\rho & = & \rho(x) \\[1em]
[\![\lambda x.E]\!]\rho & = & \langle(\lambda d : D.\ \mathsf{case}\ [\![E]\!]\rho[x \mapsto d]\ \mathsf{of} \\
 & & \qquad\qquad\quad \mid \mathsf{error} \to \mathsf{error} \\
 & & \qquad\qquad\quad \mid \langle v,t\rangle \to \langle v,t\rangle),\mathsf{tr}\rangle \\[1em]
[\![EF]\!]\rho & = & \ \ \mathsf{case}\ [\![E]\!]\rho\ \mathsf{of} \\
 & & \ \ \mid \mathsf{error} \to \mathsf{error} \\
 & & \ \ \mid \langle v,t\rangle \to\ \ \mathsf{case}\ v([\![F]\!]\rho)\ \mathsf{of} \\
 & & \qquad\qquad\qquad \mid \mathsf{error} \to \mathsf{error} \\
 & & \qquad\qquad\qquad \mid \langle v',t'\rangle \to \langle v', t \sqcup t'\rangle \\[1em]
[\![\mathsf{trust}\ E]\!]\rho & = & \ \ \mathsf{case}\ [\![E]\!]\rho\ \mathsf{of} \\
 & & \ \ \mid \mathsf{error} \to \mathsf{error} \\
 & & \ \ \mid \langle v,t\rangle \to \langle v,\mathsf{tr}\rangle \\[1em]
[\![\mathsf{distrust}\ E]\!]\rho & = & \ \ \mathsf{case}\ [\![E]\!]\rho\ \mathsf{of} \\
 & & \ \ \mid \mathsf{error} \to \mathsf{error} \\
 & & \ \ \mid \langle v,t\rangle \to \langle v,\mathsf{dis}\rangle \\[1em]
[\![\mathsf{check}\ E]\!]\rho & = & \ \ \mathsf{case}\ [\![E]\!]\rho\ \mathsf{of} \\
 & & \ \ \mid \mathsf{error} \to \mathsf{error} \\
 & & \ \ \mid \langle v,\mathsf{tr}\rangle \to \langle v,\mathsf{tr}\rangle \\
 & & \ \ \mid \langle v,\mathsf{dis}\rangle \to \mathsf{error}.
\end{array}
$$

Figure 3.5: Semantic equations.

$$\begin{array}{lcl} u, v, w & ::= & \mathsf{dis} \mid \mathsf{tr} \\ \tau, \sigma & ::= & t^u \\ s, t & ::= & \mathsf{base} \mid \tau \to \sigma. \end{array}$$

Figure 3.6: Syntax of trust-types

between tr and dis, namely: $\mathsf{tr} \leq \mathsf{dis}$. We denote by $\sqcup$ the least upper bound according to the last ordering.

The semantic equations are given in Figure 3.5. We employ a pattern matching case construct in the semantic description language.

Lemma 3.8 (Environment) *For expressions E and F and environment ρ we have:*

$$[\![E[F/x]]\!]\rho = [\![E]\!]\rho[x \mapsto [\![F]\!]\rho].$$

Proof. By structural induction on E. □

The connection between the operational calculus and the denotational semantics is the following soundness theorem.

Theorem 3.9 (Semantic Soundness) *The denotation of an expression is invariant under reduction: If $E \to F$ then $[\![E]\!] = [\![F]\!]$.*

Proof. By induction on the derivation of $E \to F$, applying the equational theory of the semantic description language and using Lemma 3.8 in the application case. □

3.3 The Type System

Our annotated type system is based on Curry's monomorphic type system for the λ-calculus, also known as *simply typed* λ-calculus. Figure 3.6 shows the mutually recursive definition of the syntax of our types. Recall from Section 3.1.1 that tr means that the value is trusted and dis means that it is untrusted.

We write u, v or w (and primed and subscripted versions thereof) for trust annotations, s and t for bare types without their outermost annotation, and σ or τ for annotated types. The generic term "type" will be used both for bare types without their outermost annotation and for annotated types.

Just as was the case in the denotational semantics, trust annotations are subject to a partial ordering $\leq$, defined as the least partial ordering including the relation $\mathsf{tr} \leq \mathsf{dis}$. This ordering is extended to bare types and annotated types such that two bare base-types are ordered only if they identical; for annotated types we have:

$$t^u \leq s^v \text{ if and only if } u \leq v \text{ and } t \leq s,$$

and for bare arrow types we define:

$$\tau \to \sigma \leq \tau' \to \sigma' \text{ if and only if } \tau' \leq \tau \text{ and } \sigma \leq \sigma',$$

$$A \vdash x : \tau \text{ if } x \in \text{dom}(A) \text{ and } A(x) = \tau \quad \text{(Var)}$$

$$\frac{A \vdash E : \tau \quad \tau \leq \tau'}{A \vdash E : \tau'} \quad \text{(Sub)}$$

$$\frac{A[x \mapsto \tau] \vdash E_1 : \sigma}{A \vdash \lambda x.E_1 : (\tau \to \sigma)^{\mathsf{tr}}} \quad \text{(Lambda)}$$

$$\frac{A \vdash E_1 : (\tau \to t^u)^w \quad A \vdash E_2 : \tau}{A \vdash E_1 E_2 : t^{u \sqcup w}} \quad \text{(App)}$$

$$\frac{A \vdash E_1 : t^u}{A \vdash \mathsf{trust}\ E_1 : t^{\mathsf{tr}}} \quad \text{(Trust)}$$

$$\frac{A \vdash E_1 : t^u}{A \vdash \mathsf{distrust}\ E_1 : t^{\mathsf{dis}}} \quad \text{(Distrust)}$$

$$\frac{A \vdash E_1 : t^{\mathsf{tr}}}{A \vdash \mathsf{check}\ E_1 : t^{\mathsf{tr}}} \quad \text{(Check)}$$

Figure 3.7: The type system.

so argument types are ordered contravariantly. This is inspired by the work on structural subtyping by Mitchell [Mit84], Fuh and Mishra [FM90], Cardelli [Car84] and others.

As in the denotational semantics we denote by $\sqcup$ the least upper bound operation on the trust lattice according to the $\leq$ ordering. In Section 3.5 we discuss several extensions of the type system to cope with recursion, modules, polymorphism and more general lattices of annotations.

3.3.1 Rules

A type assumption A is a partial function which takes a program identifier to an annotated type τ. Figure 3.7 shows the inference rules for the type system. A type judgment $A \vdash E : \tau$ means that from the assumptions A we can deduce that the expression E has type τ.

The rule for variables and the subtyping rule should give no surprises. Since lambda abstractions in the program are written by the programmer, we treat them as trusted in the type system. This only means that the *function* is trusted, and does not indicate how its argument and result are treated. In Section 3.5 we discuss how this can be extended to a larger scale setting with multiple modules written by multiple programmers.

In the application rule, the annotated type of the actual argument is required to match the annotated type of the formal argument. This includes the trustworthiness. The trust of the result of the application is the least upper bound of the result-trust from the arrow type and the trust of the function itself. The intuition is that if we cannot trust the function, we cannot trust the result of applying it.

The three rules for trust, distrust and check show that they behave as the identity on

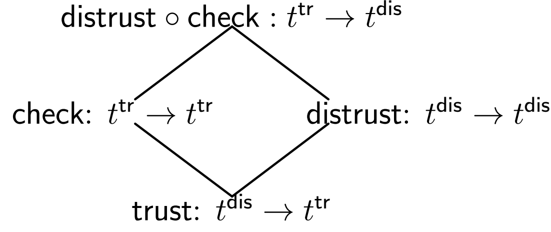


Figure 3.8: The relationship between arrow types

the underlying type. Trust makes any value trusted and distrust makes any value untrusted. Check E has a type only if E is trusted. This means that we cannot type improper normal forms and together with Subject Reduction (Theorem 3.11) this ensures the soundness of the type system.

Figure 3.8 shows the ordering of arrow types and how the constructs trust, distrust and check would fit into it.

3.3.2 Subject Reduction

An important part in proving the Subject Reduction theorem is that replacing an appropriately typed term for a variable in an expression does not change the type of the expression.

Lemma 3.10 (Substitution) *If $A[x \mapsto \sigma] \vdash E : \tau$ and $A \vdash F : \sigma$ then $A \vdash E[F/x] : \tau$.*

Proof. By induction on the derivation of $A[x \mapsto \sigma] \vdash E : \tau$. □

The main result of this section is the Subject Reduction theorem. The theorem states that types are invariant under reduction.

Theorem 3.11 (Subject Reduction) *If $A \vdash E : \tau$ and $E \to F$ then $A \vdash F : \tau$.*

Proof. By induction on the structure of the derivation of $E \to F$ and by cases on the structure of E. The (Reflex) case is trivial. In the (Sub) cases the result follows directly from the induction hypothesis. The type rule (Sub) is applicable in all cases, so when reasoning "backwards" (as in "when α E has type τ then E *must* have type σ") we must take care to handle the case where the (Sub) type rule was used in between.

For the contraction rules we show just two illustrative cases, the remaining cases are extremely similar. Suppose the last rule used in the derivation of $E \to F$ was

$$\frac{E_1 \to \mathsf{trust}\ F_1}{E = \mathsf{trust}\ E_1 \to \mathsf{trust}\ F_1 = F}\text{(Trust Contraction)}$$

By assumption we have $A \vdash \mathsf{trust}\ E_1 : t^u$ so by the rules we must have $A \vdash E_1 : t_1^{u_1}$ where $t_1 \leq t$. By the induction hypothesis we now get $A \vdash \mathsf{trust}\ F_1 : t_1^{u_1}$ and again we must have $A \vdash F_1 : t_2^{u_2}$ where $t_2 \leq t_1$. Now by the (Trust) rule of the type system we get $A \vdash \mathsf{trust}\ F_1 : t_2^{\mathsf{tr}}$ and finally by (Sub) we get $A \vdash \mathsf{trust}\ F_1 : t^u$ as required.

Another case: Suppose the last rule used in the derivation of $E \to F$ was

$$\frac{E_1 \to \mathsf{distrust}\ F_1}{E = \mathsf{distrust}\ E_1 \to \mathsf{distrust}\ F_1 = F}\text{(Distrust Contraction)}$$

$$A \vdash_C x : t \text{ if } x \in \text{dom}(A) \text{ and } A(x) = t$$

$$\frac{A[x \mapsto s] \vdash_C E_1 : t}{A \vdash_C \lambda x.E_1 : s \longrightarrow t} \qquad \frac{A \vdash_C E_1 : s \longrightarrow t \quad A \vdash_C E_2 : s}{A \vdash_C E_1 E_2 : t}$$

Figure 3.9: The Curry type system.

By assumption we know $A \vdash \mathsf{distrust}\ E_1 : t^u$ and therefore $u = \mathsf{dis}$, so by the rules we must have $A \vdash E_1 : t_1^{u_1}$ where $t_1 \leq t$. From the induction hypothesis we get $A \vdash \mathsf{distrust}\ F_1 : t_1^{u_1}$. By the (Distrust) rule we must have $A \vdash F_1 : t_2^{u_2}$ where $t_2 \leq t_1$ and $u_1 = \mathsf{dis}$. By the (Distrust) rule we now get $A \vdash \mathsf{distrust}\ F_1 : t_2^{\mathsf{dis}}$ which via (Sub) yields the required result.

Regarding application: If $E = E_1E_2 \to F_1F_2 = F$ then the result follows by two applications of the induction hypothesis. If the last rule used in the derivation of $E \to F$ was

$$\frac{E_1 \to F_1 \quad E_2 \to F_2}{E = (\mathsf{distrust}(\lambda x.E_1))E_2 \to \mathsf{distrust}\ F_1[F_2/x] = F}\text{(Application)}$$

then by assumption $A \vdash (\mathsf{distrust}(\lambda x.E_1))E_2 : t^u$. By definition of the type rules this must mean that $A \vdash \mathsf{distrust}(\lambda x.E_1) : (\sigma \to s^v)^w$ and $A \vdash E_2 : \sigma$ where $s \leq t$ and $v \sqcup w \leq u$. Again, we must also have $A \vdash \lambda x.E_1 : (\sigma_1 \to s_1^{v_1})^{w_1}$ and it must be case that $w = \mathsf{dis}$ and thus $u = \mathsf{dis}$. Also $s_1 \leq s$, $v_1 \leq v$, and $\sigma \leq \sigma_1$. Once again by the type rules we must have $A[x \mapsto \sigma_2] \vdash E_1 : s_2^{v_2}$ where $\sigma_1 \leq \sigma_2$, $s_2 \leq s_1$ and $v_2 \leq v_1$. By the (Sub) rule we get $A \vdash E_2 : \sigma_2$.

We can now apply the induction hypothesis to get $A[x \mapsto \sigma_2] \vdash F_1 : s_2^{v_2}$ and $A \vdash F_2 : \sigma_2$. By the Substitution lemma (Lemma 3.10) we then get $A \vdash F_1[F_2/x] : s_2^{v_2}$. By the (Distrust) rule we get $A \vdash \mathsf{distrust}\ F_1[F_2/x] : s_2^{\mathsf{dis}}$ and by (Sub) we get the desired result as $s_2 \leq s_1 \leq s \leq t$.

The case without $\mathsf{distrust}$ is similar. □

3.3.3 Comparison with the Curry System

Our type system may be viewed as a restriction of the classic Curry type system for λ-calculus. This notion is formalized in the following. Define the *erasure* $|\cdot|$ of a term as:

$$\begin{array}{lll@{\qquad}lll} |x| & = & x & |\lambda x.E| & = & \lambda x.|E| \\ |E_1E_2| & = & |E_1||E_2| & |\mathsf{trust}\ E| & = & |E| \\ |\mathsf{distrust}\ E| & = & |E| & |\mathsf{check}\ E| & = & |E| \end{array}$$

and likewise the erasure of an annotated type as:

$$|\mathsf{base}^u| = \mathsf{base} \qquad |(\sigma \to \tau)^w| = |\sigma| \longrightarrow |\tau|.$$

The notion of erasure is extended pointwise to environments: $|A|(x) = |t|$ if and only if $|A(x)| = t$.

The Curry type rules for erased expressions are defined in Figure 3.9. Here type assumptions A map program identifiers to Curry types.

Lemma 3.12 (Erasure) *If σ and τ are annotated types and $\sigma \leq \tau$ then $|\sigma| = |\tau|$.*

Proof. For base-types $s^u \leq t^v$ implies $s = t$. For arrow types note that by definition of $\leq$, σ and τ must have the same arrow structure. So the result follows by an induction on the common structure of σ and τ. □

Theorem 3.13 *If $A \vdash E : \tau$ then $|A| \vdash_C |E| : |\tau|$.*

Proof. By induction on the derivation of $A \vdash E : \tau$.

$E = x$: By assumption we have $x \in \mathrm{dom}(A)$ and $\tau = A(x)$. Thus $x \in \mathrm{dom}(|A|)$ and $|A|(x) = |\tau|$.

$E = \alpha\ E_1$: Here $\alpha \in \{\mathsf{trust}, \mathsf{distrust}, \mathsf{check}\}$. By the definition of erasure, $|E| = |E_1|$ and the result follows from the induction hypothesis.

$E = \lambda x.E_1$: By assumption we must have $A[x \mapsto \sigma] \vdash E_1 : \sigma'$ where $(\sigma \to \sigma')^{\mathsf{tr}} \leq \tau$. By the induction hypothesis $|A|[x \mapsto |\sigma|] \vdash_C |E_1| : |\sigma'|$. By the lambda rule in the Curry system we get $|A| \vdash_C \lambda x.|E_1| : |\sigma| \longrightarrow |\sigma'|$. By definition of erasure and the Erasure lemma we get the desired result.

$E = E_1 E_2$: By assumption we must have $A \vdash E_1 : (\sigma \to t^u)^w$ and $A \vdash E_2 : \sigma$ where $t^{u \sqcup w} \leq \tau$. From the induction hypothesis we get $|A| \vdash_C |E_1| : |\sigma| \longrightarrow |t^u|$ and $|A| \vdash_C E_2 : |\sigma|$. By the application rule in the Curry system we get $|A| \vdash_C |E_1 E_2| : |t^u|$. Finally by the Erasure lemma we get the desired result.

□

The preceding theorem also implies that the erasure of a trust-typable program is trust-typeable, but the reverse does not hold in general.

If there are no sub-terms of the form $\mathsf{check}\ E$ in a program and the erasure of the program is Curry typable then the program is trust-typable and all the trusts may be chosen as dis. This idea is formalized below. Define the *decoration* of a Curry type as

$$\begin{aligned} \Delta(\mathsf{base}) &= \mathsf{base} \\ \Delta(s \longrightarrow t) &= \Delta s^{\mathsf{dis}} \to \Delta t^{\mathsf{dis}} \end{aligned}$$

Extend decorations to Curry type assumptions: $(\Delta A)(x) = \langle \Delta t, \mathsf{dis} \rangle$ whenever $A(x) = t$. Clearly $|\Delta\ t| = t$.

Theorem 3.14 *If $A \vdash_C |E| : t$ and E contains no sub-term of the form* $\mathsf{check}\ E'$ *then $\Delta A \vdash E : \Delta t^{\mathsf{dis}}$.*

Proof. By induction on the structure of E.

$E = x$: By assumption we have $x \in \mathrm{dom}(A)$ and $A(x) = t$, so $x \in \mathrm{dom}(\Delta A)$ and $(\Delta A)(x) = (\Delta t^{\mathsf{dis}}$. By the (Var) rule we get $\Delta A \vdash x : \Delta t^{\mathsf{dis}}$.

$E = \lambda x.E_1$: By the assumptions and the Curry rules we must have $A[x \mapsto s_1] \vdash_C |E_1| : t_1$ where $s_1 \longrightarrow t_1 = t$. By induction $\Delta(A[x \mapsto s_1]) \vdash E_1 : \Delta t_1^{\mathsf{dis}}$. By the (Lambda) and (Sub) rules of the trust system, we get $\Delta A \vdash \lambda x.E_1 : (\Delta s_1^{\mathsf{dis}} \to \Delta t_1^{\mathsf{dis}})^{\mathsf{dis}}$ as required.

$$E \to E \quad \text{(Reflex)} \qquad \frac{E \to E'}{\lambda x.E \;\to\; \lambda x.E'} \text{ (Sub)}$$

$$\frac{E \to E' \qquad F \to F'}{\begin{array}{lcl} EF & \to & E'F' \\ (\lambda x.E)F & \to & E'[F'/x] \end{array}} \text{ (Application)}$$

Figure 3.10: Reductions in the ordinary λ-calculus.

$E = E_1E_2$: By the assumptions and the Curry rules we must have $A \vdash_C |E_1| : s \longrightarrow t$ and $A \vdash_C |E_2| : s$. By the induction hypothesis we get $\Delta A \vdash E_1 : (\Delta(s \longrightarrow t))^{\mathsf{dis}}$ and $\Delta A \vdash E_2 : (\Delta s)^{\mathsf{dis}}$. By the (App) rule we then get $\Delta A \vdash E_1E_2 : (\Delta t)^{\mathsf{dis}}$ as required.

$E = \mathsf{trust}\ E_1$: Since $|\mathsf{trust}\ E_1| = |E_1|$ and $A \vdash_C |E_1| : t$ by the induction hypothesis one gets $\Delta A \vdash E_1 : (\Delta t)^{\mathsf{dis}}$. Now we may apply the (Trust) and (Sub) rules to get $\Delta A \vdash \mathsf{trust}\ E_1 : (\Delta t)^{\mathsf{dis}}$ as wanted. The case for distrust is similar.

This exhausts the possible cases since E was assumed check-free. □

3.3.4 Simulation

The aim of this section is to show that for well-typed terms one may erase all the trust, distrust and check constructs and reduce expressions according to the ordinary λ-calculus as displayed in Figure 3.10 (this is taken from Definition 3.2.3 in [Bar81].) We use the same symbol for this reduction relation as for our own and it will be clear from the context which reduction relation is meant. Note that the relation defined in Figure 3.10 is a sub-relation of the reduction relation defined in Figure 3.2.

More formally the two following simulation theorems show that for well-typed terms, reduction and erasure commute: $|\cdot| \circ \to^* \;=\; \to^* \circ |\cdot|$.

In terms of implementation this means that after type-checking, an interpreter may erase all the constructs having to do with trust, and run the program without them, thus no run-time performance penalty is paid.

Lemma 3.15 (Step) *If $E \to^* \alpha\ F$ (α may be empty) and there is a reduction rule*

$$\frac{E_1 \to \alpha\ F_1}{\beta\ E_1 \to \gamma\ F_1}$$

then $\beta\ E \to^* \gamma\ F$.

Proof. By induction on the length of the sequence $E \to^* \alpha\ F$. If $E = \alpha\ F$ then by (Reflex) we have $E \to \alpha\ F$ and we may apply the rule to get $\beta\ E \to \gamma\ F$ and since $\to \subseteq \to^*$ this is the required result.

Otherwise the last step in the reduction sequence $E \to^* \alpha\ F$ must look like $E' \to \alpha\ F$, where $E \to^* E'$ and $E' \neq \alpha\ F$. Now we apply the rule mentioned in the statement of the

lemma:

$$\frac{E' \to \alpha\ F}{\beta\ E' \to \gamma\ F}$$

By the induction hypothesis one gets (via the (Sub) rule and using β for γ): $E \to^* E'$ implies $\beta\ E \to^* \beta\ E'$. By appending the two reductions we get $\beta\ E \to^* \gamma\ F$ as we wanted. In effect we get this derived rule:

$$\frac{E_1 \to^* \alpha\ F_1}{\beta\ E_1 \to^* \gamma\ F_1}$$

Similarly, from

$$\frac{E \to G \quad F \to H}{EF \to GH} \qquad \text{we get} \qquad \frac{E \to^* G \quad F \to^* H}{EF \to^* GH}$$

□

Some notation: We write $E_0 = \mathsf{CTD}^* F$ to mean that E_0 is produced by the following grammar, where F is an ordinary term.

$$E_0 ::= \mathsf{check}\ E_0 \mid \mathsf{trust}\ E_0 \mid \mathsf{distrust}\ E_0 \mid F$$

We also write $\mathsf{distrust}^?\ E$ to mean either E or $\mathsf{distrust}\ E$.

Lemma 3.16 (CTD) *If* $E = \mathsf{CTD}^*(\lambda x.E_1)$, $A \vdash E : \tau$ *and* $E_1 \to^* F_1$ *then*

$$E \to^* \mathsf{distrust}^?\ (\lambda x.F_1).$$

Proof. By induction on the length of the CTD sequence. Suppose that

$$\mathsf{CTD}^* \lambda x.E_1 = (\alpha\ (\beta\ \ldots (\lambda x.E_1) \ldots)).$$

In the base case (the empty sequence) $E_1 \to^* F_1$ implies (via Sub and Step) that $\lambda x.E_1 \to^* \lambda x.F_1$.

Otherwise, there are two cases depending on whether $(\beta\ \ldots)$ reduces to a lambda or a distrusted lambda.

Suppose that $(\beta\ \ldots) \to^* \lambda x.F_1$ by the induction hypothesis then via the Step lemma and (Lambda Contraction):

$\alpha =$ trust: $(\mathsf{trust}\ (\beta\ \ldots)) \to^* \lambda x.F_1$.

$\alpha =$ $\mathsf{distrust}$: $(\mathsf{distrust}\ (\beta\ \ldots)) \to^* \mathsf{distrust}\ \lambda x.F_1$.

$\alpha =$ check: $(\mathsf{check}\ (\beta\ \ldots)) \to^* \lambda x.F_1$.

Finally, suppose that $(\beta\ \ldots) \to^* \mathsf{distrust}\ \lambda x.F_1$ by the induction hypothesis then via the Step lemma and (Distrust Contraction):

$\alpha =$ trust: $(\mathsf{trust}\ (\beta\ \ldots)) \to^* \mathsf{trust}\ \lambda x.F_1$ and via a (Lambda Contraction) step: $\mathsf{trust}\ \lambda x.F_1 \to \lambda x.F_1$.

$\alpha =$ $\mathsf{distrust}$: $(\mathsf{distrust}\ (\beta\ \ldots)) \to^* \mathsf{distrust}\ \lambda x.F_1$.

$\alpha =$ check: As E is well-typed this case cannot occur since check(distrust E_1) is untypable.

□

Theorem 3.17 (Simulation 1) *If $A \vdash E : \tau$ and $|E| \to F$ then there is a term G such that $E \to^* G$ and $|G| = F$. Graphically:*

$$\begin{array}{ccc} E & \overset{\to^*}{\dashrightarrow} & G \\ {\scriptstyle |\cdot|}\downarrow & & \downarrow{\scriptstyle |\cdot|} \\ |E| & \overset{\to}{\longrightarrow} & F \end{array}$$

Proof. By structural induction on E.

$E =$ x: Here $|E| = E$ and the only applicable rule is (Reflex), thus we get $E = F = G$.

$E =$ $\alpha\ E_1$, where $\alpha \in \{\text{trust}, \text{distrust}, \text{check}\}$. Here we have $|E| = |E_1|$, $A \vdash E_1 : \tau'$ and $|E| = |E_1| \to F$. So by the induction hypothesis there is a G_1 such that $E_1 \to^* G_1$ and $|G_1| = F$. By the Step lemma we can deduce:

$$\frac{E_1 \to^* G_1}{E = \alpha\ E_1 \to^* \alpha\ G_1 = G}\text{(Sub)}$$

and the erasure of G is F as required.

$E =$ $\lambda x.E_1$: By the assumptions we must have $A[x \mapsto \sigma] \vdash E_1 : \sigma'$. Also, $|E| = \lambda x.|E_1|$ and $F = \lambda x.F_1$. By the nature of the reduction rules, we must have $|E_1| \to F_1$. By the induction hypothesis we know there is a G_1 such that $E_1 \to^* G_1$ and $|G_1| = F_1$. By the (Sub) rule and the Step lemma we get

$$\frac{E_1 \to^* G_1}{E = \lambda x.E_1 \to^* \lambda x.G_1 = G}\text{(Sub)}$$

and $|G| = \lambda x.|G_1| = \lambda x.F_1 = F$ as required.

$E =$ E_1E_2: By the assumptions E is well-typed thus E_1 and E_2 are well-typed. By definition of the reduction rules we must have $|E_1| \to F_1$ and $|E_2| \to F_2$. By the induction hypothesis we get G_1 and G_2 such that $E_1 \to^* G_1$, $E_2 \to^* G_2$, $|G_1| = F_1$ and $|G_2| = F_2$.

There are two cases depending on the form of $|E|$:

$|E| =$ *not a β-redex*: Here $F = F_1F_2$ where $|E_1| \to F_1$ and $|E_2| \to F_2$ so by the reasoning above $|G_1G_2| = F$ and we are done.

$|E| =$ $(\lambda x.H_1)H_2$: If $F = (\lambda x.F_1')F_2$ where $H_1 \to F_1'$ and $H_2 \to F_2$ then also $|E_1| = \lambda x.H_1 \to \lambda x.F_1' = F_1$ by (Sub). By the above statements and the Step lemma we get $E \to^* G_1G_2$ and $|G_1G_2| = F$.

Otherwise a β-reduction happens. Here $H_1 \to F_1'$, $H_2 \to F_2$ and $F = F_1'[F_2/x]$. Clearly, E_1 must have form $\mathsf{CTD}^*(\lambda x.Q_1)$ where $|Q_1| = H_1$. By the induction hypothesis there is a Q_1' such that $Q_1 \to^* Q_1'$ and $|Q_1'| = F_1'$.

By the CTD lemma $E_1 \to^* \mathsf{distrust}^? (\lambda x.Q_1')$ and by the Subject Reduction theorem (Theorem 3.11) we get that $\mathsf{distrust}^? (\lambda x.Q_1')$ is well-typed.
We may now reason as follows:

$$\frac{E_1 \to^* \mathsf{distrust}^? (\lambda x.Q_1') \quad E_2 \to^* G_2}{E_1 E_2 \to^* (\mathsf{distrust}^? (\lambda x.Q_1'))G_2} (\text{Sub + Step})$$

and

$$\frac{Q_1' \to Q_1' \quad G_2 \to G_2}{(\mathsf{distrust}^? (\lambda x.Q_1'))G_2 \to \mathsf{distrust}^? Q_1'[G_2/x]} (\text{Application})$$

since $|Q_1'| = F_1'$ and $|G_2| = F_2$:

$$|\mathsf{distrust}^? Q_1'[G_2/x]| = |Q_1'[G_2/x]| = F_1'[F_2/x] = F$$

as required.

This concludes the proof of the Simulation theorem. □

Theorem 3.18 (Simulation 2) *If $E \to F$ then $|E| \to |F|$.*

Proof. By induction on the derivation of $E \to F$.

- If $E \to F$ by (Reflex) then $|E| = |F|$ and the result holds trivially.
- If $E \to F$ by the (Sub) rule. The subterm(s) E_i of E then must reduce $E_i \to F_i$ and by the induction hypothesis $|E_i| \to |F_i|$. We may now apply the (Sub) rule to these erased terms and get $|E| \to |F|$.
- Let $\alpha, \beta, \gamma \in \{\mathsf{trust}, \mathsf{distrust}, \mathsf{check}\}$. If the last rule in the derivation of $E \to F$ was

$$\frac{E_1 \to \alpha\ F_1}{E = \beta\ E_1 \to \gamma\ F_1 = F} (\alpha\text{-Contraction})$$

  then $|E| = |E_1|$ and $|F| = |F_1|$. By the induction hypothesis we know $|E_1| \to |F_1|$ which is the desired result.
- If the last rule used in the derivation of $E \to F$ was

$$\frac{E_1 \to \lambda x.F_1}{E = \alpha\ E_1 \to \lambda x.F_1 = F} (\text{Lambda Contraction})$$

  where $\alpha \in \{\mathsf{trust}, \mathsf{check}\}$ then by the induction hypothesis we get $|E_1| \to \lambda x.|F_1|$ and since $|E| = |E_1|$ and $|F| = \lambda x.|F_1|$ this is the desired result.
- If the last rule used in the derivation of $E \to F$ was

$$\frac{E_1 \to F_1 \quad E_2 \to F_2}{E = (\mathsf{distrust}\ \lambda x.E_1)E_2 \to \mathsf{distrust}\ F_1[F_2/x] = F} (\text{Application})$$

  We have $|E| = (\lambda x.|E_1|)|E_2|$ and $|F| = |F_1|[|F_2|/x]$. By the induction hypothesis $|E_1| \to |F_1|$ and $|E_2| \to |F_2|$. We may now apply the (Application) rule to get the desired result. The case for the trusted lambda is similar.

□

3.3.5 Strong Normalization

In Curry typed λ-calculus all typable terms are strongly normalizing, i.e. they have no infinite reduction paths. This result carries over to our language, essentially since our type system admits fewer terms than the Curry system. On the other hand we have terms not occurring in the standard λ-calculus and a lot more reduction rules so this requires a proof.

Theorem 3.19 (Strong Normalization) *If $A \vdash E : \tau$ then there are no infinite reduction paths starting from E.*

Proof. By Theorem 3.13, $A \vdash E : \tau$ implies $|A| \vdash_C |E| : |\tau|$. By the Strong Normalization theorem for Curry typed λ-calculus (see for example [GTL89]) there are no infinite Curry reduction paths starting from $|E|$, and there is a unique Curry normal form F such that $|E| \to^* F$. We can now apply the first Simulation theorem (Theorem 3.17) to obtain a term G such that $E \to^* G$ and $|G| = F$.

As $|G| = F$ is a normal form it has no β-redexes, but in G some other reductions may be applicable. Because of the Church-Rosser theorem, only using contraction and Sub rules will not block any possible other reduction that might be applicable, so let $G \to^* G'$ be such a reduction sequence. In all the contraction rules the number of $\{\mathsf{check}, \mathsf{trust}, \mathsf{distrust}\}$ constructs decreases by one. Writing $\to^!$ for the *significant* part of the $\to$ relation we have that in the sequence $G \to^! G_1 \to^! \cdots \to^! G'$ there are a finite number of steps (less than the size of G). Thus we may take G' such that in G' no more of these non-β reductions are applicable.

After reaching G' can it be the case that the contractions revealed a β-redex in G'? Suppose for a contradiction that this is the case. This means that G' has a subterm of the form $(\mathsf{distrust}^? \ \lambda x.G_1)G_2$. So we have $G' \to G''$ by reducing this redex. By Theorem 3.18 this means that $F = |G'| \to^! |G''|$. But this contradicts the fact the F could be reduced no further! So it cannot be the case that the final contractions reveal a β-redex, and thus G' is a normal form. □

3.4 Type Inference

The type inference problem is:

> Given an untyped program E possibly with trust, distrust, and check expressions in it, is E typable? If so, annotate it.

From Theorem 3.13 we have that trust typing implies Curry typing. Our type inference algorithm works by first checking if the program has a Curry type and then checking a condition that only involves trust values.

3.4.1 Constraints

The type inference problem can be rephrased in terms of solving a system of constraints.

Definition 3.20 Given two disjoint denumerable sets of variables $\mathcal{V}_y$ and $\mathcal{V}_r$, a *T-system* is a pair (C, D) where:

- C is a finite set of inequalities $X \leq X'$ between constraint expressions, where X and X' are of the forms V or $V_1^{W_1} \to V_2^{W_2}$, and where $V_1, V_2 \in \mathcal{V}_y$ and $W_1, W_2 \in \mathcal{V}_r$.
- D is a finite set of constraint of the forms $W \leq W'$, $W = \mathsf{tr}$, or $W = \mathsf{dis}$, where $W, W' \in \mathcal{V}_r$.

A *solution* for a T-system is a pair of maps (δ, φ), where δ maps variables in $\mathcal{V}_y$ to types without their outermost annotation, and where φ maps variables in $\mathcal{V}_r$ to trusts, such that all constraints are satisfied. If φ satisfies all constraints in D, we say that D has solution φ. □

Given a λ-term E, assume that E has been α-converted so that all bound variables are distinct. Let $\mathcal{V}_y$ be the set consisting of:

- A variable $[\![F]\!]_y$ for each occurrence of a subterm F of E; and
- A variable x_y for each λ-variable x occurring in E.

The notation $[\![F]\!]_y$ is ambiguous because there may be more than one occurrence of F in E. However, it will always be clear from context which occurrence is meant. Intuitively, $[\![F]\!]_y$ denotes the type of F after the use of subsumption. Moreover, x_y denotes the type assigned to the bound variable x.

Let $\mathcal{V}_r$ be the set consisting of:

- A variable $[\![F]\!]_r$ for each occurrence of a subterm F of E; and
- A variable x_r for each λ-variable x occurring in E.
- A variable $\langle GH \rangle_r$ for each occurrence of an application GH in E.

As before, the notation $[\![F]\!]_r$ is ambiguous. Intuitively, $[\![F]\!]_r$ denotes the trust value of F after the use of subsumption. Moreover, x_r denotes the trust value assigned to the bound variable x. Finally, $\langle GH \rangle_r$ denotes the trust value of GH before the use of subsumption.

From the λ-term E, we generate the T-system (C, D) where:

For each occurrence in E	We have in C	We have in D
x	$x_y \leq [\![x]\!]_y$	$x_r \leq [\![x]\!]_r$
$\lambda x.F$	$x_y^{x_r} \to [\![F]\!]_y^{[\![F]\!]_r} \leq [\![\lambda x.F]\!]_y$	
GH	$[\![G]\!]_y \leq [\![H]\!]_y^{[\![H]\!]_r} \to [\![GH]\!]_y^{\langle GH \rangle_r}$	$[\![G]\!]_r \leq [\![GH]\!]_r$ $\langle GH \rangle_r \leq [\![GH]\!]_r$
$\mathsf{trust}\ F$	$[\![F]\!]_y \leq [\![\mathsf{trust}\ F]\!]_y$	
$\mathsf{distrust}\ F$	$[\![F]\!]_y \leq [\![\mathsf{distrust}\ F]\!]_y$	$[\![\mathsf{distrust}\ F]\!]_r = \mathsf{dis}$
$\mathsf{check}\ F$	$[\![F]\!]_y \leq [\![\mathsf{check}\ F]\!]_y$	$[\![F]\!]_r = \mathsf{tr}$

Denote by $T(E)$ the T-system of constraints generated from E in this fashion. The solutions of $T(E)$ correspond to the possible type annotations of E in a sense made precise by Theorem 3.23.

Let A be a trust-type environment. If δ is a function assigning types to variables in $\mathcal{V}_y$ and φ a function assigning trusts to variables in $\mathcal{V}_r$, we say that (δ, φ) *extend* A if for every x in the domain of A, we have $A(x) = \delta(x_y)^{\varphi(x_r)}$.

As a shorthand in the following, we write $(\delta, \varphi) \models (C, D)$ to mean that (δ, φ) is a solution to the constraints (C, D). Define also, for two functions δ and δ' agreeing on $\mathrm{dom}(\delta) \cap \mathrm{dom}(\delta')$, $\delta + \delta'$ as the unique function on $\mathrm{dom}(\delta) \cup \mathrm{dom}(\delta')$ that agrees with the two functions on their respective domains.

Lemma 3.21 (Soundness) *If* $(\delta, \varphi) \models T(E)$, *and* δ, φ *extend* A *then* $A \vdash E : \delta(\llbracket E \rrbracket_y)^{\varphi(\llbracket E \rrbracket_r)}$.

Proof. By induction on the structure of E. □

Lemma 3.22 (Completeness) *If* $A \vdash E : t^u$ *then there is a solution* $(\delta, \varphi) \models T(E)$ *with* δ *and* φ *extending* A, *and* $\delta(\llbracket E \rrbracket_y) = t$ *and* $\varphi(\llbracket E \rrbracket_r) = u$.

Proof. By induction on the derivation of $A \vdash E : t^u$.

$E = x$: As $A \vdash x : t^u$ we must have $x \in \mathrm{dom}(A)$, $A(x) = s^v$ and $s \leq t, v \leq u$. In this case $T(E) = \{x_y \leq \llbracket x \rrbracket_y, x_r \leq \llbracket x \rrbracket_r\}$. Put $\delta(x_y) = s$ and $\varphi(x_r) = v$ so that (δ, φ) extends A. Finally assign $\delta(\llbracket x \rrbracket_y) = t$ and $\varphi(\llbracket x \rrbracket_r) = u$ to satisfy the constraints.

$E = \lambda x.F$: By the type rules we must have $A[x \mapsto \sigma] \vdash F : s^v$ where $\sigma \to s^v \leq t$. By the induction hypothesis we get $(\delta, \varphi) \models T(F)$, $\delta(\llbracket F \rrbracket_y) = s$, $\varphi(\llbracket F \rrbracket_r) = v$, and (δ, φ) extends $A[x \mapsto \sigma]$. Now assign $\delta' = \delta[\llbracket \lambda x.F \rrbracket_y \mapsto t]$ and $\varphi' = \varphi[\llbracket \lambda x.F \rrbracket_r \mapsto u]$. Now check that $\sigma \to s^v \leq t$ implies

$$\delta'(x_y)^{\varphi'(x_r)} \to \delta'(\llbracket F \rrbracket_y)^{\varphi'(\llbracket F \rrbracket_r)} \leq \delta'(\llbracket \lambda x.F \rrbracket_y)$$

as required. So we get $(\delta', \varphi') \models T(E)$.

$E = GH$: By the type rules we must have $A \vdash G : (\sigma \to s^v)^w$, $A \vdash H : \sigma$ where $s \leq t$ and $v \sqcup w \leq u$. By the induction hypothesis we get $(\delta, \varphi) \models T(G)$ and $(\delta', \varphi') \models T(H)$ and both solutions extending A which means that they agree on their common domain (the x_y's and the x_r's in $\mathrm{dom}(A)$). The definition of $T(E)$ says

$$\begin{aligned} T(E) = T(G) \cup T(H) \cup (&\{\llbracket G \rrbracket_y \leq \llbracket H \rrbracket_y^{\llbracket H \rrbracket_r} \to \llbracket GH \rrbracket_y^{\langle GH \rangle_r}\}, \\ &\{\llbracket G \rrbracket_r \leq \llbracket GH \rrbracket_r,\ \langle GH \rangle_r \leq \llbracket GH \rrbracket_r\}) \ . \end{aligned}$$

Define $\delta'' = \delta + \delta'[\llbracket GH \rrbracket_y \mapsto t]$ and $\varphi'' = \varphi + \varphi'[\llbracket GH \rrbracket_r \mapsto u, \langle GH \rangle_r \mapsto v]$. Now, $(\delta'', \varphi'') \models T(E)$ because

$$\delta(\llbracket G \rrbracket_y) \leq \sigma \to t^v = \delta'(\llbracket H \rrbracket_y)^{\varphi'(\llbracket H \rrbracket_r)} \to \delta''(\llbracket GH \rrbracket_y)^{\varphi''(\langle GH \rangle_r)}$$

$$\begin{aligned} \varphi(\llbracket G \rrbracket_r) = w &\leq u = \varphi''(\llbracket GH \rrbracket_r) \\ \varphi''(\langle GH \rangle_r) = v &\leq u = \varphi''(\llbracket GH \rrbracket_r) \ . \end{aligned}$$

and clearly (δ'', φ'') extend A.

$E =$ check F: From the type rules we must have $A \vdash F : s^{\mathsf{tr}}$ where $s \leq t$. By the induction hypothesis we get $(\delta, \varphi) \models T(F)$, $\delta([\![F]\!]_y) = s$, and $\varphi([\![F]\!]_r) = \mathsf{tr}$. Now, $T(E) = T(F) \cup (\{[\![F]\!]_y \leq [\![\mathsf{check}\ F]\!]_y\}, \{[\![F]\!]_r = \mathsf{tr}\})$. Put $\delta' = \delta[[\![\mathsf{check}\ F]\!]_y \mapsto t]$ and $\varphi' = \varphi[[\![\mathsf{check}\ F]\!]_r \mapsto u]$.

Clearly, δ', φ' extend A, $\delta'([\![F]\!]_y) \leq \delta'([\![\mathsf{check}\ F]\!]_y)$, and $\varphi'([\![F]\!]_r) = \mathsf{tr}$ as required. The cases for trust and distrust are very similar.

□

Theorem 3.23 *The judgment $A \vdash E : t^u$ is derivable if and only if there exists a solution (δ, φ) of $T(E)$ with (δ, φ) extending A such that $\delta([\![E]\!]_y) = t$ and $\varphi([\![E]\!]_r) = u$. In particular, if E is closed, then E is typable with type t and trust u if and only if there exists a solution (δ, φ) of $T(E)$ such that $\delta([\![E]\!]_y) = t$ and $\varphi([\![E]\!]_r) = u$.*

Proof. Combine Lemma 3.21 and 3.22. □

3.4.2 Algorithm

Definition 3.24 Given a T-system (C, D), define the *deductive closure* $(\bar{C}, \bar{D})$ to be the smallest T-system such that:

- $C \subseteq \bar{C}$.
- $D \subseteq \bar{D}$.
- If $V_1^{W_1} \to V_2^{W_2} \leq V_3^{W_3} \to V_4^{W_4}$ is in $\bar{C}$, then $V_3 \leq V_1$ and $V_2 \leq V_4$ are in $\bar{C}$, and $W_3 \leq W_1$ and $W_2 \leq W_4$ are in $\bar{D}$.
- If $X_1 \leq X_2$ and $X_2 \leq X_3$ are in $\bar{C}$, then $X_1 \leq X_3$ is in $\bar{C}$.

□

Lemma 3.25 *(C, D) and $(\bar{C}, \bar{D})$ have the same solutions.*

Proof. Since $C \subseteq \bar{C}$ and $D \subseteq \bar{D}$, any solution of $(\bar{C}, \bar{D})$ is also a solution of (C, D). The converse can be proved by induction on the construction of $(\bar{C}, \bar{D})$. □

If we remove all mentioning of trust and subtyping from the type rules in Figure 3.7 and from the constraints defined earlier in this section, we obtain two equivalent formulations of Curry typability [PS95]. Clearly, E is Curry typable if and only if $|E|$ is Curry typable. The constraint system (written out below) that expresses Curry typability will be denoted Curry(E).

For each occurrence in E	We have in Curry(E)
x	$x_y = [\![x]\!]_y$
$\lambda x.F$	$[\![\lambda x.F]\!]_y = x_y \to [\![F]\!]_y$
GH	$[\![G]\!]_y = [\![H]\!]_y \to [\![GH]\!]_y$
trust F	$[\![\mathsf{trust}\ F]\!]_y = [\![F]\!]_y$
distrust F	$[\![\mathsf{distrust}\ F]\!]_y = [\![F]\!]_y$
check F	$[\![\mathsf{check}\ F]\!]_y = [\![F]\!]_y$

To aid the definition of our type inference algorithm we define the following operations: If s, t are bare trust types such that $|s| = |t|$, then define the operators $\sqcup^{|s|}$ and $\sqcap^{|t|}$ as follows.

$$s \sqcup^{|s|} t = \begin{cases} \mathsf{base} & \text{if } s = t = \mathsf{base} \\ (s_1 \sqcap^{|s_1|} t_1)^{u_1 \sqcap v_1} \to (s_2 \sqcup^{|s_2|} t_2)^{u_2 \sqcup v_2} & \text{if } s = s_1^{u_1} \to s_2^{u_2} \\ & \text{and } t = t_1^{v_1} \to t_2^{v_2} \end{cases}$$

$$s \sqcap^{|s|} t = \begin{cases} \mathsf{base} & \text{if } s = t = \mathsf{base} \\ (s_1 \sqcup^{|s_1|} t_1)^{u_1 \sqcup v_1} \to (s_2 \sqcap^{|s_2|} t_2)^{u_2 \sqcap v_2} & \text{if } s = s_1^{u_1} \to s_2^{u_2} \\ & \text{and } t = t_1^{v_1} \to t_2^{v_2} \end{cases} .$$

If $t_1, \ldots, t_n$ are bare trust types, and s is a Curry type such that $|t_i| = s$ for all $i \in 1..n$, then define $\bigsqcup_i^s t_i = t_1 \sqcup^s \ldots \sqcup^s t_n$. If t is a Curry type, define

$$\begin{array}{rclrcl} \mathsf{small}(\mathsf{base}) & = & \mathsf{base} & \mathsf{big}(\mathsf{base}) & = & \mathsf{base} \\ \mathsf{small}(s \to t) & = & \mathsf{big}(s)^{\mathsf{dis}} \to \mathsf{small}(t)^{\mathsf{tr}} & \mathsf{big}(s \to t) & = & \mathsf{small}(s)^{\mathsf{tr}} \to \mathsf{big}(t)^{\mathsf{dis}} \end{array}$$

If s is a bare trust type and t is a Curry type such that $|s| = t$, then $s \sqcup^t \mathsf{small}(t) = s$ and $s \sqcap^t \mathsf{big}(t) = s$. In other words, $\mathsf{small}(t)$ is the *least* bare type with erasure t, and $\mathsf{big}(t)$ is the greatest such.

For each constraint expression X define

$$L(C, X) = \{V_1^{W_1} \to V_2^{W_2} \mid V_1^{W_1} \to V_2^{W_2} \leq X \text{ is in } \bar{C}\} .$$

Intuitively, $L(C, X)$ is the set of syntactic lower bounds for X.

We also define the erasure of a constraint expression used in C, mapping trust-type constraint expressions to Curry constraint expressions:

$$\begin{array}{rcl} |V| & = & V \\ |V_1^{W_1} \to V_2^{W_2}| & = & V_1 \to V_2 \end{array}$$

where $V_1, V_2 \in \mathcal{V}_y$ and $W_1, W_2 \in \mathcal{V}_r$.

Lemma 3.26 *If $T(E) = (C, D)$, and ψ is a solution to* $\mathsf{Curry}(E)$*, and $X_1 \leq X_2$ is a constraint in $\bar{C}$, then $\psi(|X_1|) = \psi(|X_2|)$.*

Proof. By induction on the construction of $\bar{C}$. □

Theorem 3.27 *Suppose $T(E) = (C, D)$. Then $T(E)$ is solvable if and only if E is Curry typable and $\bar{D}$ is solvable.*

Proof. Suppose first that $T(E)$ is solvable. By Theorem 3.23, E is trust typable. It follows from Theorem 3.13 and the remark above that E is Curry typable, and from Lemma 3.25 that $\bar{D}$ is solvable.

For the reverse implication, suppose that $\mathsf{Curry}(E)$ has solution ψ and that $\bar{D}$ has solution φ. We define δ inductively in the Curry types of the constraint variables.

$$\begin{aligned}\delta(V) = {} & \textsf{if } \psi(|V|) = \textsf{base then base} \\ & \quad \textsf{else let } \{V_{1i}^{W_{1i}} \to V_{2i}^{W_{2i}}\} = L(C,V) \cup \{\textsf{small}(\psi(|V|))\} \\ & \quad\quad \textsf{in } \bigsqcup_i^{\psi(|V|)} (\delta(V_{1i})^{\varphi(W_{1i})} \to \delta(V_{2i})^{\varphi(W_{2i})})\end{aligned}$$

To see that δ is well-defined, we need that the Curry types of the variables V_{1i} and V_{2i} are of strictly less size than the Curry type of V. For $(V_{1i}^{W_{1i}} \to V_{2i}^{W_{2i}}) \in L(C,V)$, we get by Lemma 3.26 that $\psi(|V_{1i}^{W_{1i}} \to V_{2i}^{W_{2i}}|) = \psi(|V|) = s \to t$ for some s,t. This means that $\psi(|V_{1i}|) = s$ and $\psi(|V_{2i}|) = t$ which are both of smaller size than $s \to t$, so δ is well-defined.

To see that (δ, φ) is a solution of $T(E)$, consider an inequality $X_1 \leq X_2$ in C. If $\psi(|X_1|) = \textsf{base}$, then by Lemma 3.26, $\psi(|X_2|) = \textsf{base}$, $\delta(X_1) = \delta(X_2) = \textsf{base}$, thus $\delta(X_1) \leq \delta(X_2)$ as required.

In case $\psi(|X_1|) = s \to t$, we have by Lemma 3.26 that $\psi(|X_2|) = s \to t$ and since $\bar{C}$ is transitively closed we get $L(C,X_1) \subseteq L(C,X_2)$ so $\delta(X_1) \leq \delta(X_2)$ as required. □

Using the characterization of Theorem 3.27, we get a type inference algorithm:

Input: A λ-term E of size n.
1: Construct $T(E) = (C,D)$ (in log space).
2: Close (C,D), yielding $(\bar{C},\bar{D})$ (in $O(n^3)$ time, see for example [Pal95]).
3: Check if E is Curry typable (in $O(n)$ time).
4: Check if $\bar{D}$ is solvable (in $O(n^2)$ time).
5: If E is Curry typable and $\bar{D}$ is solvable,
then output "typable"
else output "not typable".

The entire algorithm requires $O(n^3)$ time. To construct an annotation of a typable program, we can use the construction of the second half of the proof of Theorem 3.27.

3.5 Extensions

In this section we discuss several possible extensions of the type system.

Recursion. The type system can be extended to handle recursion by adding a rec rule. In the (untyped) reduction system, the rec combinator can be coded with the classical Y combinator: $\textsf{rec } x.E \equiv \textsf{Y}(\lambda x.E)$. The following reduction rule is a derived rule in the λ-calculus and in our system, and correspondingly we would have a rec rule in the type system:

$$\frac{E \to^* F}{\textsf{Y}E \to^* F(\textsf{Y}E)} \qquad\qquad \frac{A[x \mapsto \tau] \vdash E_1 : \tau}{A \vdash \textsf{rec } x.E_1 : \tau}$$

Subject Reduction still holds, but Strong Normalization of course fails in this case. The type inference algorithm can also be extended in a straightforward way to deal with the rec construct.

Polymorphism. ML style let polymorphism can be achieved in the usual way by replacing let bound variables with their definition. This is of course inefficient as each definition might then be type-checked many times. The type system can be extended along the same ideas that extend Curry types to Hindley-Milner types. An extension of our type inference algorithm remains to be found.

Finding a good type inference algorithm for a type-system with both structural subtyping and polymorphism is a nontrivial task although the work by Aiken and Wimmers [AW93] and by Eifrig, Smith and Trifonov [EST95] is promising.

A Trust-case Construction. One could imagine the usefulness of a trust-case construction that would allow dynamic dispatch on the trustworthiness of a value. The reduction rules added for such a construction could be:

$$\frac{E \to \mathsf{trust}\ E'}{\mathsf{trust_case}\ E\ F\ G \to F(\mathsf{trust}\ E')} \qquad \frac{E \to \lambda x.E'}{\mathsf{trust_case}\ E\ F\ G \to F(\lambda x.E')}$$

$$\frac{E \to \mathsf{distrust}\ E'}{\mathsf{trust_case}\ E\ F\ G \to G(\mathsf{distrust}\ E')}$$

and the corresponding type-rule:

$$\frac{A \vdash E : t^u \quad A \vdash F : t^{\mathsf{tr}} \to \tau \quad A \vdash G : t^{\mathsf{dis}} \to \tau}{A \vdash \mathsf{trust_case}\ E\ F\ G : \tau}$$

Church-Rosser and the Subject Reduction theorem still holds with these extensions and generating constraints for this construct is not hard either. However, this would only make sense in the presence of a polymorphic trust type system. With monomorphic trust-types all the trust-case choices would be statically determinable from the type system, so such a construction would be of very limited use. And since we have not developed an inference algorithm for a polymorphic trust type system, this has not been an issue.

Other Lattices. The values of trust-tags may be extended from the two point lattice used in this paper to any finite lattice. Extending the lattice to a longer linear lattice accommodates multiple levels of trust. Extensions to non-linear orderings may allow different properties to be modeled at once: Take the four point lattice $(\mathcal{P}(\{\mathsf{path_ok}, \mathsf{signature_ok}\}), \subseteq)$ with the empty set denoting completely untrusted. This could be used in a web server that can both verify digital signatures and do consistency checking on URL paths. In such a situation one would extend check to a construct checking for reverse subset inclusion.

Modules. In a larger scale system with many program modules and many programmers, it is useful to differentiate between functions located in different modules such that there would be trusted and untrusted modules, where functions defined in untrusted modules would not be trusted in any other module. This can be realized in our simple system by having a preprocessor that wraps all lambdas in an untrusted module in the distrust construct. Some external programming environment might also be used to ensure that only trustworthy programmers get to write trusted modules.

One might also make another distinction among modules, akin to the difference between safe and unsafe modules in Modula-3 [CDG+89], where only unsafe modules are allowed to use arbitrary type casts and unlimited address arithmetic. In a trust analysis system, unsafe modules would then correspond to modules where the trust construct is used, and just as in Modula-3 one has to take extra care in the unsafe modules.

3.6 Related Work

The original notion of trust analysis was presented in [Ørb95], Chapter 2 of this thesis, where an abstract interpretation analysis and a constraint based analysis for an imperative, first order language with pointers were given. This work extends trust analysis to the higher order functional case and formalizes it in terms of an annotated type system.

In [Mit84] Mitchell developed the structural subtyping idea and our type system borrows some of these ideas to handle automatic coercion from trusted data to untrusted data.

In an earlier version of the paper we used a different syntax for trust-types inspired by the work on effect systems by for example Gifford and Jouvelot, writing for example $\mathsf{Bool} \stackrel{\mathsf{tr}\ \mathsf{dis}}{\longrightarrow} \mathsf{Bool}\ \#\ \mathsf{tr}$ for what is now written $(\mathsf{Bool}^{\mathsf{tr}} \to \mathsf{Bool}^{\mathsf{dis}})^{\mathsf{tr}}$. This turned out to be misleading in that our type system does not involve accumulating representations of side effects and input/output. We thank our referees for pointing this out and making us change the syntax of types.

3.7 Summary

We have argued for the usefulness of so-called trust analysis to help programmers produce safer and more trustworthy software. We have presented an extension of the λ-calculus together with a reduction semantics as well as a sound denotational semantics. The reduction calculus is proved Church-Rosser. We then gave a type system that enables the static inference of the trustworthiness of values and the type system was proved to have the Subject Reduction property with respect to the semantics of our language.

We have related our extension of the λ-calculus to the classical λ-calculus and obtained two simulation theorems, as well as shown that well-typed terms in our calculus are strongly normalizing.

Then a constraint based type inference algorithm was presented and proved correct with respect to the type system.

Finally we have discussed certain possible extensions to our analysis and given several examples of why it is different from already known analyses.

3.7.1 Acknowledgements

Part of this chapter was completed while both authors were visiting the Laboratory for Computer Science at the Massachusetts Institute of Technology. We want to thank the anonymous referees for many valuable comments and Philip Wadler for encouragement.

Chapter 4

Dependence Algebra

4.1 Introduction

The notion of *data dependence* is a central concept in program analysis. It is is used in optimizing compilers to determine which statements can be re-ordered to speed up computation [ASD86], it is used in binding-time analysis to determine whether an expression depends on dynamic data. It is used in security flow analysis to make sure that secret data does not end up in insecure places [Den76, Den82, BBM94]. Dependences are also important in program slicing (a program slice being the relevant part of the program with respect to some variables) [Wei84, Agr94, Tip95]. Data dependences are also of central importance in the work on value dependence graphs [WCES94], which provided some inspiration for the work described here.

Most program analyses treat data dependences as boolean: a variable is either dependent on another variable or it is not. That is, the dependences between variables form a relation. This chapter extends the notion of dependence from two-valued to many-valued: a variable can be more or less dependent on another variable.

We treat dependences as the primary objects, and introduce algebraic operations on dependences. The following chapters exploit the theory developed here to give a trust analysis for C code, and a soft type inference analysis for action semantics specifications, respectively.

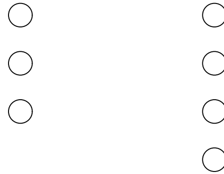
The following sections attempt to make our notion of dependence precise, and develop an algebra of dependences that is used to calculate dependences in a program from the program text, often in a compositional manner.

4.1.1 Slots

A dependence is a connection between two things: a variable may depend on another variable, an expression may depend on a set of variables, and a statement may depend on some preceding statements, and so on. Before we can define our notion of dependence we must tie down these things that our dependences exist between.

We will define dependences to exist between *slots*. A slot is a place that can hold a value, for example a variable, a parameter position, an array element, or the location holding the result of a sub-expression. We distinguish between input- and output-slots. A program accepts inputs in some slots and produces outputs in some possibly different slots. An arithmetic expression, for example, has as its input slots the free variables of the expression, and a single output slot for the result.

Slots may be mutable in the case of an imperative language or immutable in the case of pure functional languages. A mutable slot is a slot that occurs both as an input slot and as an output slot. We will picture a set of slots as a column of circles, input slots on the left and output slots on the right:

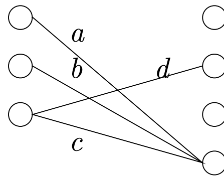


4.1.2 The Meaning of Dependence

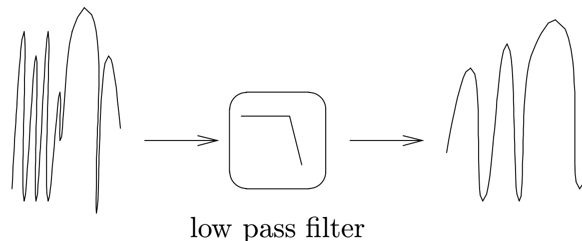
Traditional program analyses treat data dependences as boolean: a slot is either dependent or independent of another slot; there is only one kind of dependence. This does not contradict that most analyses operate with the notion that a slot *may* (or may not) depend on a set of other slots. A slot is either included in the set of other slots, or it is not.

We generalize this boolean notion of dependence in order to put more information into the dependence and distinguish different kinds of dependences.

Dependences will be pictured as labeled edges between slots, with missing edges denoting independence. In may be useful to think of each dependence edge as a *channel* that values may flow along from one slot to another. The label on each edge denotes the "capacity" of the channel.



Another analogy is to think of a dependence edge as a low pass filter as it is used in electronics and signal processing:



low pass filter

The label on the edge determines the "cut off frequency" of the dependence.

The following defines the general concept of a dependence algebra. An element of the carrier set of the algebra is a representation of a particular kind of dependence between two slots. The algebra has three operations: join ($\sqcup$), sum ($+$), and sequential composition ($\cdot$). The join operation models combination of dependences along optional execution paths, such as the branches of a conditional. Consider for example the program:

if P then
  $x := y + 2$

```
else
  x := string-append(y, "foo")
end if
```

where x depends on y both as a number and as a string.

We require the join operator to be commutative (the ordering of conditional branches makes no difference), idempotent (all the necessary approximation is made in one step), and associative (the order of approximation is not essential). These requirements make the join operation behave like a real least upper bound in a join-semilattice. The semi-lattice ordering is induced by the operator in the usual way: $a \sqcup b = b \iff a \sqsubseteq b$.

In some of the applications of this technical machinery there is no intuitive notion of a least (bottom) element of the semi-lattice ordering, we do therefore not require the existence of a least element in the semi-lattice. The lack of a bottom element allows the non-existence of fixed points of monotone endo-functions on the structure. And even when a fixed point exists, it may not be found by iteration from below, as there is no unique place to start. In practice we can exploit the fixed point approximation technique described in Sect. 4.2.1. In the cases where the semi-lattice is finite and has a least element it is easy to see that it is a full lattice.

The sum operator models combination of dependences along parallel execution paths. If y depends on x through d_1 along one thread of execution and through d_2 along another thread of execution then after the threads join, y depends on x through $d_1 + d_2$. Consider the following example:

```
(y,z) := (x,x);
w := y * z;
```

First the parallel assignment assigns x's value to both y and z, then w is assigned the product of y and z. The dependence of w on x is computed as the sum of the dependences via y and z.

We require the sum to be commutative and associative for the same reasons that the join was required to have these properties. When a value flows along two parallel channels, it has to be able to pass through each channel unharmed, it therefore makes no difference whether there is one channel or two parallel channels of the same capacity. We shall thus require the sum to be idempotent. The sum has a natural zero element encoding the non-existence of a channel between two slots. For technical reasons we shall also require the sum to be monotone with respect to the ordering induced by the join operation.

The multiplication operation $(\cdot)$ models sequential composition of computations, as in the example:

```
x := y;
z := x
```

Here the sequential composition of the two assignments makes z depend on y via the intermediate variable x. We require the multiplication to be associative, have a unit element corresponding to a channel allowing anything to pass, and have the non-existence of a channel as a zero element. As for the sum we require that the multiplication is monotone with respect to the join-ordering. We shall also require that multiplication distributes over sums: adding

two parallel channels and then appending a third channel with capacity c to the end of them is the same as appending channels of capacity c to each of the two parallel channels.

The following definitions formalize the notion of an algebra of dependences between individual slots. This is later used to build descriptions of dependences among several slots.

4.2 Basic Definitions

Definition 4.1 A *dependence algebra* (a **DAlg**-structure) $(C, \sqcup, +, \cdot, \mathbf{0}, \mathbf{1})$ is an algebra with carrier set C, three binary operations $(+, \sqcup, \cdot)$, and distinguished elements: $\mathbf{0}, \mathbf{1} \in C$; such that the axioms below are satisfied. For $a, b, c \in C$:

- $a \sqcup b = b \sqcup a$ (commutative).
- $a \sqcup a = a$ (idempotent).
- $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$ (associative).

These axioms determine the join semi-lattice ordering $\sqsubseteq$.

- $a + b = b + a$ (commutative).
- $a + a = a$ (idempotent).
- $a + (b + c) = (a + b) + c$ (associative).
- $\mathbf{0} + a = a$ (neutral element).
- $a \sqsubseteq b \Rightarrow a + c \sqsubseteq b + c$ (monotone).

The operator $+$ is also monotone in its second argument by commutativity.

The third operator, $\cdot$, will be written as juxtaposition when no confusion can arise. It satisfies the following axioms:

- $(ab)c = a(bc)$ (associative).
- $\mathbf{1}a = a\mathbf{1} = a$ (unit).
- $\mathbf{0}a = a\mathbf{0} = \mathbf{0}$ (annihilator).
- If $a \sqsubseteq b$ then $ac \sqsubseteq bc$ and $ca \sqsubseteq cb$ (monotone).
- $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$ (distributive).

□

Note that the axioms for $+$ can also be interpreted as specifying a least upper bound operation on a semi-lattice with $\mathbf{0}$ as its bottom. We shall, however, not pursue this view further.

The following well-known fact will be used several times in the following sections exploiting the monotonicity of the $+$ and $\cdot$ operations.

Fact 4.2 *The endo-function f on a join semi-lattice is monotone if and only if*

$$\bigsqcup_i f(x_i) \sqsubseteq f(\bigsqcup_i x_i).$$

Proof. If f is monotone and $x_i \sqsubseteq \bigsqcup_i x_i$ we have $f(x_i) \sqsubseteq f(\bigsqcup_i x_i)$, and since the right-hand side is a common upper bound of all the $f(x_i)$, we get $\bigsqcup_i f(x_i) \sqsubseteq f(\bigsqcup_i x_i)$.

For the other direction: if $x \sqsubseteq y$ then $x \sqcup y = y$ thus $f(x \sqcup y) = f(y)$, and by assumption $f(x) \sqcup f(y) \sqsubseteq f(x \sqcup y) = f(y)$, which is equivalent to $f(x) \sqsubseteq f(y)$. □

By duality, we have that f is monotone on a meet semi-lattice iff $f(\bigsqcap_i x_i) \sqsubseteq \bigsqcap_i f(x_i)$.

Recall also (for example from [Grä78] or [Bir48]) that in a (semi-) lattice a is said to *cover* b if $b \sqsubseteq a$ and there is no element x such that $b \sqsubset x \sqsubset a$. Furthermore, a is called an *atom* if a covers the bottom element of a semi-lattice.

The following definition recalls the notion of a semi-ring [Eil74] (a ring except that the additive operation is a monoid, not a group).

Definition 4.3 A semi-ring $(C, +, \cdot, \mathbf{0}, \mathbf{1})$ consists of a carrier set C, binary operations $+$ and $\cdot$, and has two distinguished elements $\mathbf{0}, \mathbf{1} \in C$. The operations are subject to the following axioms:

- The structure $(C, +, \mathbf{0})$ forms a commutative monoid ($+$ is associative, commutative, and has $\mathbf{0}$ as a neutral element).
- The structure $(C, \cdot, \mathbf{1})$ forms a monoid.
- The zero element, $\mathbf{0}$, is an annihilator for the multiplicative operation: $\mathbf{0}a = a\mathbf{0} = \mathbf{0}$.
- Multiplication distributes over addition: $a(b + c) = ab + ac$ and $(a + b)c = ac + bc$.

□

The sub-structure $(C, +, \cdot, \mathbf{0}, \mathbf{1})$ of a **DAlg**-structure $(C, \sqcup, +, \cdot, \mathbf{0}, \mathbf{1})$ immediately satisfies the axioms of a semi-ring. We shall use this property when we define matrices and matrix multiplication over dependence algebras. As dependence algebras are just semi-rings with an associated (semi-lattice) ordering they could justifiably also be named *ordered semi-rings.*

Algebraic structures (groups, rings, semi-groups) with an associated ordering have been studied for many years in mathematics. Fuchs [Fuc63] sums up most of the known results until 1960, and many papers on the topic have been published in the *Semigroup Forum* journal. Most of the work has naturally centered around *total* orders and algebraic structures with more structure than semi-rings. I have been able to find precious few results applicable to the structures considered here.

4.2.1 Limits

The semi-lattice structure of a **DAlg**-structure does not necessarily have a least element. The usual fixed point iteration theorem, that says that one can find the least fixed point of a continuous function by iteratively applying the function starting from the bottom, does not apply directly anymore.

Still, while analyzing recursive programs and loops, we need to find a sound approximation of a sequence $m, f(m), f(f(m)), \ldots$ where m is an element of the semi-lattice structure, specific to the particular analysis in question, and f is a monotone endo-function on the **DAlg**-structure.

As m is not *least* in the ordering we cannot conclude that $m \sqsubseteq f(m)$, and we may therefore not deduce that the sequence $m, f(m), f(f(m)), \ldots$ is non-decreasing. This foils the fixed point iteration theorem as the sequence may not reach a single fixed point, but contain a proper cycle. In the following we construct an alternative sequence from the above sequence, show that it is non-decreasing and that it dominates that original sequence. With this we can find an upper approximation of all the elements of the original sequence.

In the following let f be a monotone endo-function on the **DAlg**-structure D, and let m be an element of D.

Let (x_i) be the sequence defined by $x_0 = m$, and $x_{i+1} = f(x_i)$. We define the sequence (y_i) by $y_0 = m$, and $y_{i+1} = f(y_i) \sqcup y_i$.

It is clear that $x_0 \sqsubseteq y_0$. If we assume that $x_i \sqsubseteq y_i$ then

$$x_{i+1} = f(x_i) \sqsubseteq f(y_i) \sqsubseteq f(y_i) \sqcup y_i = y_{i+1}.$$

By this inductive argument every element of the sequence (y_i) dominates the corresponding element in the sequence (x_i). The sequence (y_i) is non-decreasing as:

$$y_{i+1} = f(y_i) \sqcup y_i \sqsupseteq y_i.$$

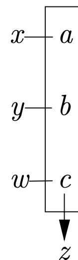
This implies that $y_n \sqsupseteq x_i$ for all $i \leq n$. As (y_i) is non-decreasing it reaches a fixed point on a **DAlg**-structure satisfying the ascending chain condition in a finite number of steps, that is, there is an n such that $y_n = f(y_n) \sqcup y_n$, and by induction: for all $i \geq n$: $y_n = y_i$. As (y_i) dominates (x_i), and (y_i) is non-decreasing we therefore get $y_n \sqsupseteq x_i$ for all i. So the limit of the (y_i) sequence is a safe approximation of all the elements of the (x_i) sequence. Note that the limit of the (y_i) sequence coincides with the limit of (x_i) in the case where m is *least*.

Join semi-lattices of finite height trivially satisfy the ascending chain condition, and our program analyses shall consider only such semi-lattices in order to be implementable.

The construction of the (y_i) sequence defines a way to compute a safe approximation in the analysis of recursive calls. The construction of the (y_i) sequence above is hardly new, and appears at least implicit in [CC79a, CC79b].

4.2.2 Vectors

Most programs have multiple input/output slots. To model a slot depending on a set of (other) slots we employ vectors of dependences and extend the three algebraic operations to vectors of dependences. The assignment "$z := x + y + w$" would give rise to the dependence vector pictured below:



We use vectors instead of mere sets of dependences because the position of the individual dependences will be important later, when we define vector concatenation. It also simplifies some of the machinery, enabling us to talk about matrix multiplication instead of "relational" composition.

A dependence vector $v = (a_1, a_2, \ldots, a_n)$ models the dependence of an output slot on the input slots numbered 1 to n, such that the output slot depends on slot number i through dependence a_i. Sums and least upper bounds of dependences are extended to vectors componentwise.

Definition 4.4 We write $(a_1, a_2, \ldots, a_n)$ for a vector of length n with elements a_i of the underlying **DAlg**-structure. The operations of sum and least upper bound are extended componentwise to vectors:

$$\begin{aligned}(a_1, a_2, \ldots, a_n) + (b_1, b_2, \ldots, b_n) &= (a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n),\\ (a_1, a_2, \ldots, a_n) \sqcup (b_1, b_2, \ldots, b_n) &= (a_1 \sqcup b_1, a_2 \sqcup b_2, \ldots, a_n \sqcup b_n).\end{aligned}$$

Multiplication of a vector by a scalar is defined as usual taking due care of the fact that the multiplication need not be commutative.

$$\begin{aligned}a(b_1, b_2, \ldots, b_n) &= (ab_1, ab_2, \ldots, ab_n),\\ (a_1, a_2, \ldots, a_n)b &= (a_1 b, a_2 b, \ldots, a_n b).\end{aligned}$$

□

A vector of length m shall be called an m-vector in the following. We shall name vectors by lower case letters $v, w, x, y, z, \ldots$

As a **DAlg**-structure is a semi-ring we can define scalar products of vectors in the usual way.

Definition 4.5 The scalar product of two vectors v and w, of equal length, is:

$$\langle v, w \rangle = \sum_i v_i w_i.$$

□

The fact below recalls that the scalar product is a tensor.

Fact 4.6 *Scalar products over a* **DAlg***-structure are bi-linear. Let x, y, and z be vectors of length n and a be a scalar, then*

$$\begin{aligned}\langle x, y + z \rangle &= \langle x, y \rangle + \langle x, z \rangle,\\ \langle x + y, z \rangle &= \langle x, z \rangle + \langle y, z \rangle,\\ \langle xa, y \rangle &= \langle x, ay \rangle,\\ \langle ax, y \rangle &= a\langle x, y \rangle,\\ \langle x, ya \rangle &= \langle x, y \rangle a.\end{aligned}$$

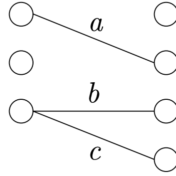
Proof. Simple calculation. □

4.2.3 Matrices

To model dependences between many input slots and many output slots we first consider matrices of dependences.

Given a **DAlg**-structure D we can consider the set of $m \times n$ matrices with entries from D. Let us call this set $M_{mn}(D)$ in the following. Starting from the operations on D we can define similar operations on $M_{mn}(D)$. We shall write capital letters $A, B, C, \ldots$ for matrices. If A is a matrix in $M_{mn}(D)$ we write A_{ij} for the entry in the i'th row and j'th column.

It is useful to think of a matrix as a bi-partite graph with input slots on the left and output slots on the right. Edges between slots denote dependences, and absent edges denote independence.



If we name the matrix pictured above A, and assuming the numbering of slots starts from the top we have $A_{11} = \mathbf{0}$, $A_{12} = a$, and so on:

$$A = \begin{pmatrix} \mathbf{0} & a & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & b & c \end{pmatrix}$$

Think of information as flowing along the edges of the graph from left to right, constrained by the capacity of the edges as denoted by the algebra elements.

Definition 4.7 The operations $\sqcup$, and $+$ on $M_{mn}(D)$ are defined componentwise as follows. Let $A, B \in M_{mn}(D)$, and i, j be row and column numbers respectively. Then we define:

$$\begin{aligned} (A \sqcup B)_{ij} &= A_{ij} \sqcup B_{ij}, \\ (A + B)_{ij} &= A_{ij} + B_{ij}. \end{aligned}$$

□

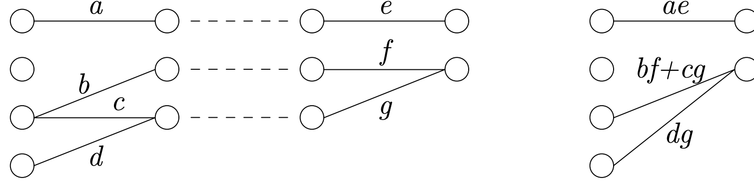
We define sequential composition as matrix multiplication.

Definition 4.8 Let A be an $m \times n$ matrix, and B be an $n \times k$ matrix. Then define for $i \leq m$ and $j \leq k$:

$$(AB)_{ij} = \sum_{1 \leq k \leq n} A_{ik} B_{kj}.$$

□

Matrix multiplication may be pictured graphically as in the figure below where the two matrices on the left are composed to give the matrix on the right.



In analyses using the dependence algebra framework developed here, we need to reason about the computational complexity of the algorithms. In the following we assume that the operations on the underlying **DAlg**-structure can be done in constant time to simplify the complexity measures. With this assumption it is easy to see that the cost of performing sums and joins of $m \times n$ matrices is in $O(mn)$, and the cost of the above matrix multiplication is in $O(mkn)$. We cannot use Strassen's algorithm for matrix multiplication as it requires the underlying structure to be a ring, which is not the case in general with **DAlg**-structures.

Theorem 4.9 *If D is a* **DAlg***-structure then $(M_{mm}(D), \sqcup, +, \cdot, \mathbf{0}, \mathbf{1})$ is a* **DAlg***-structure.*

Proof. Define $\mathbf{0} \in M_{mm}(D)$ to be the everywhere $\mathbf{0}$ matrix, and define $\mathbf{1} \in M_{mm}(D)$ to be the usual square matrix with $\mathbf{1}$ in the diagonal and $\mathbf{0}$ everywhere else. By the general theorem that matrices over a semi-ring constitute a semi-ring we get that all the axioms for a **DAlg**-structure concerning sum and product (except the monotonicity axioms) are fulfilled.

The axioms for $\sqcup$ hold on $M_{mm}(D)$ as $\sqcup$ is defined componentwise. As the $\sqsubseteq$ ordering is defined from $\sqcup$ the monotonicity of sum and product follows from their monotonicity on the underlying structure D. □

The work on information flow relations by Bergeretti and Carré [BC85] used some of same intuitions as employed here, however, only boolean matrices were considered to represent the relations between inputs and outputs, so a more fine-grained analysis is possible with the richer dependence algebras.

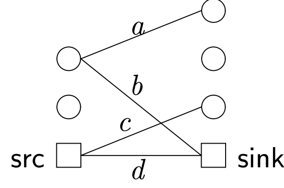
4.2.4 Quadruples

A dependence *quadruple* (called a *quad* for short) is a dependence matrix connecting input slots to output slots together with a special input slot: src, acting as the universal producer of constants, and a special output slot: sink, acting as a consumer. Quadruples will be named $Q, Q_i, Q', \ldots$ in the following.

Definition 4.10 An $m \times n$ quadruple $Q = (s, A, t, u)$ consists of a row vector s of length n, an $m \times n$ matrix A, a column vector t of length m, and a scalar u. □

The vector s contains the dependences between the special slot src and the output slots, whereas the vector t contains dependences between the input slots and the slot sink. Finally, the scalar u holds the dependence between src and sink for completeness. Below is a picture of the 2×3 quad:

$$(s, A, t, u) = ((\mathbf{0}, \mathbf{0}, c), \begin{pmatrix} a & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix}, (b, \mathbf{0}), d),$$



Note that the size of a quad may be 0 in one or both of the dimensions. A quad with zero input size has an empty matrix A, and an empty sink vector t, whereas the source vector s is of the output length. Dually, if the output size is 0 then A and s are empty, and t has the input length. Sums and least upper bounds of quads are defined componentwise.

Definition 4.11 Let $Q = (s, A, t, u)$ and $Q' = (s', A', t', u')$ be $m \times n$ quadruples. We then define the binary operations $\sqcup$, and $+$ on them as:

$$\begin{aligned}(s, A, t, u) \sqcup (s', A', t', u') &= (s \sqcup s', A \sqcup A', t \sqcup t', u \sqcup u'),\\ (s, A, t, u) + (s', A', t', u') &= (s + s', A + A', t + t', u + u'),\end{aligned}$$

We also define the constant: $\mathbf{0} = (\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0})$. □

The cost of computing the above sum and join of quads is in $O((m + 1)(n + 1))$, where we add one to each dimension to take care of the case where m or n may be zero.

The definition of sequential composition/product of quads takes care that the dependences relating the special slots src and sink are persistent, i.e., if sink already depends on x in the quad Q then composing Q with another quad doesn't change that. The same persistence holds for the src slot.

Definition 4.12 Let $Q = (s, A, t, u)$ be an $m \times n$ quad, and $Q' = (s', A', t', u')$ be an $n \times k$ quad. The product QQ' is an $m \times k$ quad defined as:

$$(s, A, t, u)(s', A', t', u') = (sA' + s', AA', t + At', st' + u + u').$$

For square quads we define the unit quad as: $\mathbf{1} = (\mathbf{0}, \mathbf{1}, \mathbf{0}, \mathbf{0})$. □

The cost of computing the quad product is dominated by the embedded matrix multiplication, and is thus in $O((m + 1)(k + 1)(n + 1))$.

We shall write $Q_{mn}(D)$ for the set of $m \times n$ quadruples over the **DAlg**-structure D. When D is understood from the context we shall write just Q_{mn} for this set.

Proposition 4.13 *Quad multiplication is associative. Let* $Q_0 = (s_0, A_0, t_0, u_0) \in Q_{mn}$, $Q_1 = (s_1, A_1, t_1, u_1) \in Q_{nk}$, *and* $Q_2 = (s_2, A_2, t_2, u_2) \in Q_{kl}$. *Then*

$$(Q_0Q_1)Q_2 = Q_0(Q_1Q_2).$$

Proof. We compute:

$$\begin{aligned}&(s_0, A_0, t_0, u_0)((s_1, A_1, t_1, u_1)(s_2, A_2, t_2, u_2))\\ &= (s_0, A_0, t_0, u_0)(s_1A_2 + s_2, A_1A_2, t_1 + A_1t_2,\\ &\qquad s_1t_2 + u_1 + u_2)\end{aligned}$$

$$
\begin{aligned}
&= (s_0A_1A_2 + s_1A_2 + s_2, A_0A_1A_2, t_0 + A_0(t_1 + A_1t_2),\\
&\quad s_0(t_1 + A_1t_2) + u_0 + s_1t_2 + u_1 + u_2)\\
&= (s_0A_1A_2 + s_1A_2 + s_2, A_0A_1A_2, t_0 + A_0t_1 + A_0A_1t_2,\\
&\quad s_0t_1 + s_0A_1t_2 + u_0 + s_1t_2 + u_1 + u_2)\\
&= (s_0A_1 + s_1, A_0A_1, t_0 + A_0t_1, s_0t_1 + u_0 + u_1)(s_2, A_2, t_2, u_2)\\
&= ((s_0, A_0, t_0, u_0)(s_1, A_1, t_1, u_1))(s_2, A_2, t_2, u_2).
\end{aligned}
$$

□

Proposition 4.14 *Quad multiplication distributes over addition. Let* $Q_0 = (s_0, A_0, t_0, u_0) \in Q_{mn}$*,* $Q_1 = (s_1, A_1, t_1, u_1) \in Q_{nk}$*,* $Q_2 = (s_2, A_2, t_2, u_2) \in Q_{nk}$*, and* $Q_3 \in Q_{kl}$*. Then*

$$
\begin{aligned}
Q_0(Q_1 + Q_2) &= Q_0Q_1 + Q_0Q_2,\\
(Q_1 + Q_2)Q_3 &= Q_1Q_3 + Q_2Q_3.
\end{aligned}
$$

Proof. Using idempotency of $+$, we prove only the first equation. The second equation follows similarly.

$$
\begin{aligned}
&(s_0, A_0, t_0, u_0)((s_1, A_1, t_1, u_1) + (s_2, A_2, t_2, u_2))\\
&= (s_0, A_0, t_0, u_0)(s_1 + s_2, A_1 + A_2, t_1 + t_2, u_1 + u_2)\\
&= (s_0(A_1 + A_2) + s_1 + s_2, A_0(A_1 + A_2), t_0 + A_0(t_1 + t_2),\\
&\quad s_0(t_1 + t_2) + u_0 + u_1 + u_2)\\
&= (s_0A_1 + s_0A_2 + s_1 + s_2, A_0A_1 + A_0A_2, t_0 + A_0t_1 + A_0t_2,\\
&\quad s_0t_1 + s_0t_2 + u_0 + u_1 + u_2)\\
&= (s_0A_1 + s_1 + s_0A_2 + s_2, A_0A_1 + A_0A_2, t_0 + A_0t_1 + t_0 + A_0t_2,\\
&\quad s_0t_1 + u_0 + u_1 + s_0t_2 + u_0 + u_2)\\
&= (s_0A_1 + s_1, A_0A_1, t_0 + A_0t_1, s_0t_1 + u_0 + u_1)\\
&\quad +(s_0A_2 + s_2, A_0A_2, t_0 + A_0t_2, s_0t_2 + u_0 + u_2)\\
&= (s_0, A_0, t_0, u_0)(s_1, A_1, t_1, u_1) + (s_0, A_0, t_0, u_0)(s_2, A_2, t_2, u_2).
\end{aligned}
$$

□

If D is a **DAlg**-structure then $(Q_{mm}(D), \sqcup, +, \cdot, \mathbf{0}, \mathbf{1})$ is almost a **DAlg**-structure. The axioms for $+$ and $\sqcup$ are easily seen to be satisfied, as the two operations are defined componentwise. Multiplication is associative by Prop. 4.13, and distributive over $+$ by Prop. 4.14.

Monotonicity of multiplication for quadruples follows from the monotonicity of sum and product for matrices, vectors and scalars.

However, $\mathbf{0}$ is no annihilator:

$$
\begin{aligned}
\mathbf{0}(s, A, t, u) &= (s, \mathbf{0}, \mathbf{0}, u),\\
(s, A, t, u)\mathbf{0} &= (\mathbf{0}, \mathbf{0}, t, u).
\end{aligned}
$$

Note that an $m \times n$ quad $Q = (s, A, t, u)$ can be represented by a $(m+2) \times (n+2)$ matrix in the following way[1]:

$$
\begin{pmatrix}
A & \mathbf{0} & t\\
s & \mathbf{1} & u\\
\mathbf{0} & \mathbf{0} & \mathbf{1}
\end{pmatrix}
$$

[1] This was remarked by an anonymous referee.

This matrix structure is stable under addition and the $\sqcup$ operation, and matrix multiplication corresponds to multiplication of quads.

4.3 Image

Given a vector of input slots it is useful to be able to "push it through" a quad obtaining a vector of output slots. This is formalized as the *image* construction.

Definition 4.15 Let v be an m-vector over D and let $Q = (s, A, t, u)$ be an $m \times n$ quad over D. The *image* of v through Q is then defined as:

$$\text{img}(v, Q) = s + vA.$$

□

It is clear from the monotonicity of $+$ and matrix multiplication that $\text{img}(v, Q)$ is monotone in both v and Q.

The following theorem shows that one can push a vector through a series of quads one at a time or compose the quads first and then push the vector through the composition. This will prove to be one of the central properties used in the soundness proofs in Chapters 5 and 6.

Theorem 4.16 *Let v be an m-vector, $Q = (s, A, t, u) \in Q_{mn}(D)$, and $Q' = (s', A', t', u') \in Q_{nk}(D)$. Then*

$$\text{img}(\text{img}(v, Q), Q') = \text{img}(v, QQ').$$

Proof.

$$\begin{aligned} \text{img}(\text{img}(v, Q), Q') &= \text{img}(s + vA, Q') \\ &= s' + (s + vA)A' \\ &= s' + sA' + vAA'. \end{aligned}$$

$$\begin{aligned} \text{img}(v, QQ') &= \text{img}(v, (sA' + s', AA', t + At', st' + u + u')) \\ &= sA' + s' + vAA'. \end{aligned}$$

□

Theorem 4.17 *Let v be an m-vector, $Q = (s, A, t, u) \in Q_{mn}(D)$, and $Q' = (s', A', t', u') \in Q_{mn}(D)$. Then*

$$\text{img}(v, Q + Q') = \text{img}(v, Q) + \text{img}(v, Q').$$

Proof.

$$\begin{aligned} \text{img}(v, Q + Q') &= \text{img}(v, (s + s', A + A', t + t', u + u')) \\ &= s + s' + v(A + A') \\ &= \text{img}(v, Q) + \text{img}(v, Q'). \end{aligned}$$

□

The rest of the material of this chapter will not be needed until Chapter 6.

4.4 Pre-image

One of the more important features of the quad abstraction of a program is that a quad can be applied “backwards” in contrast to the program function which can only be run in one direction.

This section defines the notion of a *pre-image* of a vector v through a quad, i.e., the input necessary to obtain v as an output image of the quad. The pre-image defined here does not give very accurate information, and for certain underlying **DAlg**-structures it is possible to define more accurate notions of a pre-image. An example of this is developed in Section 4.11.

Definition 4.18 Let $Q = (s, A, t, u) \in Q_{mn}$, and let v be an n-vector. The *pre-image* of v under Q is defined as:

$$\text{pre}(Q, v) = Av + t.$$

□

Just as for the image function, it is clear that pre is monotone in both its arguments.

Theorem 4.19 *Let v be a k-vector, $Q = (s, A, t, u) \in Q_{mn}$, and $Q' = (s', A', t', u') \in Q_{nk}$. Then*

$$\text{pre}(Q, \text{pre}(Q', v)) = \text{pre}(QQ', v).$$

Proof.

$$\begin{aligned}\text{pre}(Q, \text{pre}(Q', v)) &= \text{pre}(Q, A'v + t') \\ &= A(A'v + t') + t \\ &= AA'v + At' + t.\end{aligned}$$

and

$$\begin{aligned}\text{pre}(QQ', v) &= \text{pre}((sA' + s', AA', t + At', st' + u + u'), v) \\ &= AA'v + t + At'.\end{aligned}$$

□

Theorem 4.20 *Let v a n-vector, $Q = (s, A, t, u) \in Q_{mn}$, and $Q' = (s', A', t', u') \in Q_{mn}$. Then*

$$\text{pre}(Q + Q', v) = \text{pre}(Q, v) + \text{pre}(Q', v).$$

Proof.

$$\begin{aligned}\text{pre}(Q + Q', v) &= \text{pre}((s + s', A + A', t + t', u + u'), v) \\ &= (A + A')v + t + t' \\ &= \text{pre}(Q, v) + \text{pre}(Q', v).\end{aligned}$$

□

It may be useful to propagate information first in the forward direction and then backwards through the same quad, computing $\text{pre}(Q, \text{img}(v, Q))$, and dually $\text{img}(\text{pre}(Q, v), Q)$. The values of these computations are in general not comparable under the $\sqsubseteq$ ordering.

4.5 Concatenation

It is natural to concatenate vectors. This will prove very useful when we apply the theory to soft typing for action semantics where vector (tuple) concatenation is heavily used.

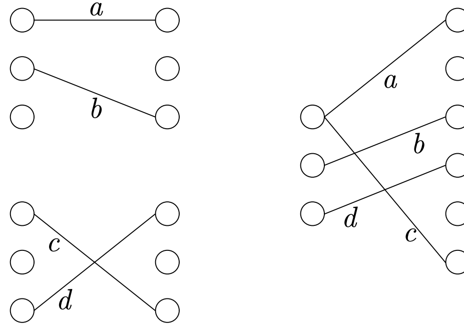
From vector concatenation we derive notions of concatenation for matrices and quads that will also prove useful in the action semantics application.

Definition 4.21 The concatenation of vectors v and w is written $v +\!\!+ w$ and is defined as:

$$(a_1, a_2, \ldots, a_n) +\!\!+ (b_1, b_2, \ldots, b_m) \quad = \quad (a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m).$$

□

From the above definition we can derive a notion of concatenation for matrices. This models the concatenation of the output slots of two matrices with the same input slots, as in the figure below where the two matrices on the left are concatenated on the right.



Definition 4.22 Let A be an $m \times n$ matrix and B an $m \times k$ matrix. Then the concatenation $A +\!\!+ B$ is an $m \times (n + k)$ matrix with entries defined as:

$$(A +\!\!+ B)_{ij} \quad = \quad \begin{cases} A_{ij} & \text{if } j \leq n, \\ B_{i(j-n)} & \text{otherwise.} \end{cases}$$

□

If we write $A_{i\cdot}$ for the i'th row vector in A we can define matrix concatenation in terms of concatenation of row vectors:

$$(A +\!\!+ B)_{i\cdot} = A_{i\cdot} +\!\!+ B_{i\cdot}.$$

The cost of matrix concatenation is easily seen to be in $O(m(n + k))$.

Concatenation is further extended to quads where special care must be taken to handle the sink vectors.

Definition 4.23 Let (s, A, t, u) be an $m \times n$ quad, and (s', A', t', u') be an $m \times k$ quad. Then their concatenation is defined as:

$$(s, A, t, u) +\!\!+ (s', A', t', u') \quad = \quad (s +\!\!+ s', A +\!\!+ A', t + t', u + u').$$

□

As for matrix concatenation, quad concatenation is in $O((m+1)(n+k+1))$, where we again add one to handle the case of "one-dimensional" quads. Matrix multiplication distributes from the left over concatenation.

Proposition 4.24 *Let A be an $m \times n$ matrix over D, $B \in M_{nk}(D)$, and $C \in M_{nk'}(D)$. Then*

$$A(B \mathbin{+\!\!+} C) = AB \mathbin{+\!\!+} AC.$$

Proof.

$$\begin{aligned}
(A(B \mathbin{+\!\!+} C))_{ij} &= \sum_r A_{ir}(B \mathbin{+\!\!+} C)_{rj} \\
&= \sum_r A_{ir} \begin{cases} B_{rj} & \text{if } j \le k \\ C_{r(j-k)} & \text{otherwise} \end{cases} \\
&= \begin{cases} \sum_r A_{ir}B_{rj} & \text{if } j \le k \\ \sum_r A_{ir}C_{r(j-k)} & \text{otherwise} \end{cases} \\
&= \begin{cases} (AB)_{ij} & \text{if } j \le k \\ (AC)_{i(j-k)} & \text{otherwise} \end{cases} \\
&= (AB \mathbin{+\!\!+} AC)_{ij}.
\end{aligned}$$

□

To prove the corresponding theorem for quads we need the following little lemma.

Lemma 4.25 *Let v, v' be n-vectors, and w, w' be m-vectors. Then*

$$(v \mathbin{+\!\!+} w) + (v' \mathbin{+\!\!+} w') = (v + v') \mathbin{+\!\!+} (w + w').$$

Proof. The vectors v and v' have the same length and the vectors w and w' have the same lengths. The $+$ operation is defined componentwise. □

Theorem 4.26 *Let $Q_1 = (s_1, A_1, t_1, u_1) \in Q_{mn}(D)$, $Q_2 = (s_2, A_2, t_2, u_2) \in Q_{nk}(D)$, and $Q_3 = (s_3, A_3, t_3, u_3) \in Q_{nk'}(D)$. Then*

$$Q_1(Q_2 \mathbin{+\!\!+} Q_3) = Q_1Q_2 \mathbin{+\!\!+} Q_1Q_3.$$

Proof. Using the previous Lemma and idempotency of $+$ we compute:

$$\begin{aligned}
Q_1(Q_2 \mathbin{+\!\!+} Q_3) &= Q_1(s_2 \mathbin{+\!\!+} s_3, A_2 \mathbin{+\!\!+} A_3, t_2 + t_3, u_2 + u_3) \\
&= (s_1(A_2 \mathbin{+\!\!+} A_3) + (s_2 \mathbin{+\!\!+} s_3), A_1(A_2 \mathbin{+\!\!+} A_3), \\
&\qquad t_1 + A_1(t_2 + t_3), s_1(t_2 + t_3) + u_1 + u_2 + u_3) \\
&= ((s_1A_2 \mathbin{+\!\!+} s_1A_3) + (s_2 \mathbin{+\!\!+} s_3), A_1A_2 \mathbin{+\!\!+} A_1A_3, \\
&\qquad t_1 + A_1t_2 + A_1t_3, s_1t_2 + s_1t_3 + u_1 + u_2 + u_3) \\
&= ((s_1A_2 + s_2) \mathbin{+\!\!+} (s_1A_3 + s_3), A_1A_2 \mathbin{+\!\!+} A_1A_3, \\
&\qquad (t_1 + A_1t_2) + (t_1 + A_1t_3), s_1t_2 + u_1 + u_2 + s_1t_3 + u_1 + u_3) \\
&= (s_1A_2 + s_2, A_1A_2, t_1 + A_1t_2, s_1t_2 + u_1 + u_2) \\
&\qquad \mathbin{+\!\!+} (s_1A_3 + s_3, A_1A_3, t_1 + A_1t_3, s_1t_3 + u_1 + u_3) \\
&= Q_1Q_2 \mathbin{+\!\!+} Q_1Q_3.
\end{aligned}$$

□

Concatenation is monotone with respect to the $\sqsubseteq$ ordering. This is obvious for vectors and matrices as the ordering is defined componentwise. For quads we state it as a proposition.

Proposition 4.27 *Let $Q_1, Q_2 \in Q_{nm}(D)$ and $Q_3 \in Q_{nk}(D)$. Then*

$$\begin{array}{rcl} Q_1 \sqsubseteq Q_2 & \Rightarrow & Q_1 +\!\!\!+ Q_3 \sqsubseteq Q_2 +\!\!\!+ Q_3, \text{ and} \\ & & Q_3 +\!\!\!+ Q_1 \sqsubseteq Q_3 +\!\!\!+ Q_2. \end{array}$$

Proof. By monotonicity of concatenation for matrices and vectors and monotonicity of $+$ for vectors and scalars. □

Concatenation and image interacts in the following way.

Theorem 4.28 *Let $Q_1 = (s_1, A_1, t_1, u_1) \in Q_{mn}$, $Q_2 = (s_2, A_2, t_2, u_2) \in Q_{mk}$, and let v be an m-vector. Then*

$$\mathrm{img}(v, Q_1 +\!\!\!+ Q_2) = \mathrm{img}(v, Q_1) +\!\!\!+ \mathrm{img}(v, Q_2).$$

Proof. Using Prop. 4.24 and Lemma 4.25 we compute:

$$\begin{array}{rcl} \mathrm{img}(v, Q_1 +\!\!\!+ Q_2) & = & v(A_1 +\!\!\!+ A_2) + (s_1 +\!\!\!+ s_2) \\ & = & (vA_1 +\!\!\!+ vA_2) + (s_1 +\!\!\!+ s_2) \\ & = & (vA_1 + s_1) +\!\!\!+ (vA_2 + s_2) \\ & = & \mathrm{img}(v, Q_1) +\!\!\!+ \mathrm{img}(v, Q_2). \end{array}$$

□

4.6 Quad Languages

A single quad is meant to abstract an expression that uses an input vector of a certain length and produces an output vector of a certain length. As such a single quad cannot model an expression that may use input vectors of varying lengths and produce outputs of varying lengths. The need to handle such functions is obvious, and the concatenation operation necessitates it as well.

We first abstract such functions into *sets* of quads, each element of the set abstracting the function for a particular input/output length. Later in this chapter we describe a method for approximating such potentially infinite sets in a finite way. Since quads may be concatenated like strings we shall call a set of quads a *quad language*. We define composition and concatenation of languages as follows:

Definition 4.29 Let L, L' be quad languages. Then

$$\begin{array}{rcl} LL' & = & \{QQ' \mid \exists m, n, k : Q \in L \cap Q_{mn} \text{ and } Q' \in L' \cap Q_{nk}\}, \\ L +\!\!\!+ L' & = & \{Q +\!\!\!+ Q' \mid \exists m, n, k : Q \in L \cap Q_{mn} \text{ and } Q' \in L' \cap Q_{mk}\}. \end{array}$$

The intersections with Q_{mn} in the definition are to ensure that only quads of compatible sizes are composed. □

Union of languages is just union of the languages as sets. We do not define notions of sum and join of languages as such notions will not be necessary in the following. If necessary such notions can be defined pointwise: $L + L' = \{Q + Q' \mid Q \in L, Q' \in L'\}$. The image of a set of vectors through a quad language is defined in the natural way:

$$\mathrm{img}(V, L) = \{\mathrm{img}(v, Q) \mid \exists n : v \in V, Q \in L \cap Q_{|v|n}\}.$$

It is natural to consider formal expressions over quad languages by introducing variables x ranging over languages, and formal expressions according to the grammar:

$$E ::= x \mid EE \mid E \mathbin{+\!\!+} E \mid E \cup E \mid L,$$

where L denotes a constant quad language. Such formal expressions are interpreted over quad languages by assigning a language to each variable and interpreting the three binary operators in the obvious way. We may furthermore consider sets of formal equations of the form $\{x_i = E_i\}$, where $i \in \{1, \ldots, n\}$. The desired solution to such a set of equations is the least (with respect to set inclusion) assignment of quad languages to the variables x_i such that the equations are satisfied[2]. Such a set of recursive equations would naturally be called a *quad grammar* by analogy with context free grammars for describing sets of strings.

Note that our grammars produce sets of quads, they are not related to two-dimensional string languages and the matrix grammars of Abraham and others [Abr65, DP89].

For dependence analyses of structured programs (without arbitrary control flow) it is natural to compositionally extract a quad grammar from the program syntax, whence the analysis consists merely of solving the resulting set of recursive equations. The derivation of a grammar from a program is inspired by the works of Reynolds [Rey69], and Jones and Muchnick [JM81, Jon87] on grammar-based analysis.

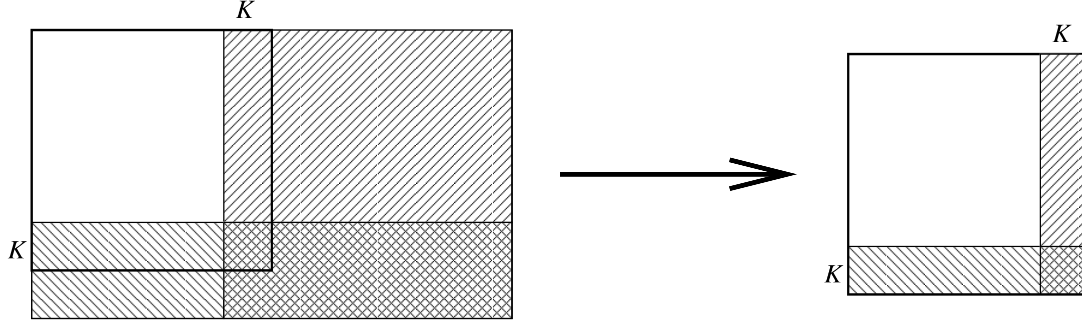
The two applications of dependence analysis in this thesis (a trust analysis for C, and a type inference for action semantics, developed in chapters 5 and 6) do not, however, exploit this simple idea. The C language allows arbitrary control flow by the `goto` statement, and our subset of action notation includes higher-order functions, requiring more advanced means of analysis.

4.7 Approximating a Single Quad

For the soft type analysis for action notation we will be generating sets of quads for each action construct to accommodate different input and output sizes.

We can approximate a single large quad by a smaller representation by collapsing rows and columns beyond a certain fixed number K into the K'th row and column:

[2] This is well-defined as all the operations on quad languages (composition, concatenation, and union) are monotone with respect to set inclusion.



This is a very simple minded approach, certainly much simpler than for example the approach by Deutsch in his thesis [Deu92] to finitely represent infinite relations between regular languages. Unfortunately we cannot directly reuse his abstraction as he is abstracting *relations* which lack both the ordering between slots (the reason for our use of vectors) and the algebra of dependences between slots.

We first define how to approximate a long vector by a shorter vector.

Definition 4.30 Let $(a_1, a_2, \ldots, a_m)$ be a vector with $m \geq K$ and define

$$\alpha_K^0(a_1, a_2, \ldots, a_m) = (a_1, a_2, \ldots, a_{K-1}, \bigsqcup_{K \leq i \leq m} a_i).$$

For $m < K$ we define $\alpha_K^0(a_1, \ldots, a_m) = (a_1, \ldots, a_m)$. □

Matrices are abstracted similarly.

Definition 4.31 Let A be an $m \times n$ matrix over D. For $i \leq m$, $i < K$ and $j \leq n$, $j < K$ define the $\alpha_K^0(A)$ to be the $\min(K, m) \times \min(K, n)$ matrix defined by:

$$\begin{aligned}
(\alpha_K^0(A))_{ij} &= A_{ij}, \\
(\alpha_K^0(A))_{Kj} &= \bigsqcup_{K \leq k \leq m} A_{kj} && \text{when } m \geq K, \\
(\alpha_K^0(A))_{iK} &= \bigsqcup_{K \leq k \leq n} A_{ik} && \text{when } n \geq K, \\
(\alpha_K^0(A))_{KK} &= \bigsqcup_{K \leq k \leq m} \bigsqcup_{K \leq l \leq n} A_{kl} && \text{when } (m, n) \geq (K, K).
\end{aligned}$$

□

The above definition uses the same K in both dimensions. It would be more general to consider a different K for each dimension, but the added generality does not buy much, one can just work with the larger of the two K's in both dimensions. Applying α_K^0 to an $m \times n$ matrix is in $O(mn)$.

We can combine the two previous definitions to define abstraction of a single quad as:

Definition 4.32 Let $Q = (s, A, t, u) \in Q_{mn}(D)$. We define the abstraction of Q as:

$$\alpha_K^0(Q) = (\alpha_K^0(s), \alpha_K^0(A), \alpha_K^0(t), u).$$

□

As for quad multiplication and concatenation we add one to the dimensions in the complexity of α_K^0 on quads: $O((m+1)(n+1))$. It is not hard to see that all three instances of the α_K^0 function are idempotent: $\alpha_K^0(\alpha_K^0(Q)) = \alpha_K^0(Q)$. Also α_K^0 is monotone.

Proposition 4.33 *The function α_K^0 distributes over $\sqcup$. Let $Q, Q' \in Q_{mn}(D)$ then*

$$\alpha_K^0(Q \sqcup Q') = \alpha_K^0(Q) \sqcup \alpha_K^0(Q').$$

Proof. Starting with the equivalent statement for vectors and using the idempotency of $\sqcup$. □

The following lemma states that adding two matrices before abstraction is more precise than first abstracting and then adding. The corresponding theorems hold for the addition of vectors and quads, but we show it for matrices only.

Lemma 4.34 *Let $A, B \in M_{mn}(D)$. Then*

$$\alpha_K^0(A+B) \sqsubseteq \alpha_K^0(A) + \alpha_K^0(B).$$

Proof. We show only one of the four cases. Assume $n \geq K$:

$$\begin{aligned} \alpha_K^0(A+B)_{iK} &= \bigsqcup_{j \geq K} (A+B)_{ij} \\ &= \bigsqcup_{j \geq K} A_{ij} + B_{ij} \\ &\sqsubseteq \bigsqcup_{j \geq K} A_{ij} + \bigsqcup_{j \geq K} B_{ij} \\ &= \alpha_K^0(A)_{iK} + \alpha_K^0(B)_{iK}. \end{aligned}$$

□

Lemma 4.35 *Let v, w be m-vectors with $m \geq K$. Then*

$$\langle v, w \rangle \sqsubseteq \langle \alpha_K^0(v), \alpha_K^0(w) \rangle.$$

Proof. Using monotonicity and idempotency of $+$ we compute:

$$\begin{aligned} \langle \alpha_K^0(v), \alpha_K^0(w) \rangle &= \sum_k \alpha_K^0(v)_k \alpha_K^0(w)_k \\ &= \sum_{k<K} v_k w_k + \sum_{k \geq K} (\bigsqcup_{j \geq K} v_j)(\bigsqcup_{j \geq K} w_j) \\ &\sqsupseteq \sum_{k<K} v_k w_k + \sum_{k \geq K} v_k w_k \\ &= \langle v, w \rangle. \end{aligned}$$

□

Proposition 4.36 *Let $A \in M_{mn}(D)$ and $B \in M_{nk}(D)$. Then*

$$\alpha_K^0(AB) \sqsubseteq \alpha_K^0(A)\alpha_K^0(B).$$

Proof. We show only the case where $k > K$ and compute:

$$\begin{aligned}
\alpha^0_K(AB)_{iK} &= \bigsqcup_{l\geq K}(AB)_{il} \\
&= \bigsqcup_{l\geq K}\sum_j A_{ij}B_{jl} \\
&\sqsubseteq \sum_j \bigsqcup_{l\geq K} A_{ij}B_{jl} \\
&= \sum_{j<K}\bigsqcup_{l\geq K} A_{ij}B_{jl} + \sum_{j\geq K}\bigsqcup_{l\geq K} A_{ij}B_{jl} \\
&\sqsubseteq \sum_{j<K} A_{ij}(\bigsqcup_{l\geq K} B_{jl}) + \sum_{j\geq K}(\bigsqcup_{h\geq K} A_{ih})(\bigsqcup_{l\geq K,h\geq K} B_{lh}) \\
&= \sum_j \alpha^0_K(A)_{ij}\alpha^0_K(B)_{jK} \\
&= (\alpha^0_K(A)\alpha^0_K(B))_{iK}.
\end{aligned}$$

The other cases (ij, Kj, and KK) are similar. □

The corresponding theorem holds for quads as well.

Theorem 4.37 *Let $Q = (s, A, t, u) \in Q_{mn}(D)$ and $Q' = (s', A', t', u') \in Q_{nk}(D)$. Then*

$$\alpha^0_K(QQ') \sqsubseteq \alpha^0_K(Q)\alpha^0_K(Q').$$

Proof. Using the previous theorems concerning the interaction of α^0_K and sums and products for vectors and matrices we can compute:

$$\begin{aligned}
\alpha^0_K(QQ') &= \alpha^0_K((sA' + s', AA', t + At', st' + u + u')) \\
&\sqsubseteq (\alpha^0_K(s)\alpha^0_K(A') + \alpha^0_K(s'), \alpha^0_K(A)\alpha^0_K(A'), \\
&\quad \alpha^0_K(t) + \alpha^0_K(A)\alpha^0_K(t'), st' + u + u') \\
&\sqsubseteq (\alpha^0_K(s)\alpha^0_K(A') + \alpha^0_K(s'), \alpha^0_K(A)\alpha^0_K(A'), \\
&\quad \alpha^0_K(t) + \alpha^0_K(A)\alpha^0_K(t'), \alpha^0_K(s)\alpha^0_K(t') + u + u') \\
&= \alpha^0_K(Q)\alpha^0_K(Q').
\end{aligned}$$

□

Using Lemma 4.34 and Prop. 4.36 we get the following.

Corollary 4.38

$$\alpha^0_K(\mathrm{img}(v, Q)) \sqsubseteq \mathrm{img}(\alpha^0_K(v), \alpha^0_K(Q)).$$

We now proceed to examine the interaction of α^0_K with concatenation.

Proposition 4.39 *Let $v = (a_1, \ldots, a_m)$ and $w = (a_{m+1}, \ldots, a_{m+n})$ be vectors over D. Then*

$$\alpha^0_K(v +\!\!+ w) = \alpha^0_K(\alpha^0_K(v) +\!\!+ \alpha^0_K(w)).$$

Proof. There are four cases: if both $m < K$ and $n < K$ then $\alpha_K^0(v) = v$ and $\alpha_K^0(w) = w$ by definition and the result is immediate. The other cases depend on the idempotency of the $\sqcup$ operator. If $m < K$ and $n \geq K$ we have:

$$\begin{aligned}
\alpha_K^0(\alpha_K^0(v) + \alpha_K^0(w)) &= \alpha_K^0(v + \alpha_K^0(w)) \\
&= \alpha_K^0(a_1, \ldots, a_m, a_{m+1}, \ldots, a_{m+K-1}, \bigsqcup_{m+K \leq i \leq m+n} a_i) \\
&= (a_1, \ldots, a_m, \ldots, a_{K-1}, \bigsqcup_{K \leq i \leq m+n} a_i) \\
&= \alpha_K^0(a_1, \ldots, a_m, \ldots, a_{m+n}) \\
&= \alpha_K^0(v + w).
\end{aligned}$$

If $m \geq K$ and $n \geq K$ we have the following:

$$\begin{aligned}
&\alpha_K^0(\alpha_K^0(v) + \alpha_K^0(w)) \\
&= \alpha_K^0(a_1, \ldots, a_{K-1}, (\bigsqcup_{K \leq i \leq m} a_i), a_{m+1}, \ldots, a_{m+K-1}, \bigsqcup_{m+K \leq i \leq m+n} a_i) \\
&= (a_1, \ldots, a_{K-1}, \bigsqcup_{K \leq i \leq m+n} a_i) \\
&= \alpha_K^0(v + w).
\end{aligned}$$

The case with $m \geq K$ and $n < K$ is similar. □

The same result holds for concatenation of matrices, but first we need a lemma.

Lemma 4.40 *Let $A \in M_{mn}(D)$ and $B \in M_{mk}(D)$. Then*

$$\bigsqcup_{i \geq K} (A + B)_{i\cdot} = (\bigsqcup_{i \geq K} A_{i\cdot}) + (\bigsqcup_{i \geq K} B_{i\cdot}).$$

Proof. Using that $\sqcup$ is defined componentwise for vectors we can compute:

$$\begin{aligned}
\bigsqcup_{i \geq K} (A + B)_{i\cdot} &= \bigsqcup_{i \geq K} A_{i\cdot} + B_{i\cdot} \\
&= (\bigsqcup_{i \geq K} A_{i\cdot}) + (\bigsqcup_{i \geq K} B_{i\cdot}).
\end{aligned}$$

□

Proposition 4.41 *Let $A \in M_{mn}(D)$ and $B \in M_{mk}$. Then*

$$\alpha_K^0(A + B) = \alpha_K^0(\alpha_K^0(A) + \alpha_K^0(B)).$$

Proof. For $i < K$ we have $\alpha_K^0(A_{i\cdot}) = \alpha_K^0(A)_{i\cdot}$ so we may use the theorem for vectors to get:

$$\begin{aligned}\alpha_K^0((A + B)_{i\cdot}) &= \alpha_K^0(A_{i\cdot} + B_{i\cdot}) \\ &= \alpha_K^0(\alpha_K^0(A_{i\cdot}) + \alpha_K^0(B_{i\cdot})).\end{aligned}$$

For the remaining case ($i \geq K$) we apply the preceding lemma and the fact:

$$\bigsqcup_{i\geq K} \alpha_K^0(A)_{i\cdot} = \alpha_K^0(A)_{K\cdot} = \alpha_K^0(\bigsqcup_{i\geq K} A_{i\cdot}),$$

and compute:

$$\begin{aligned}\alpha_K^0(\alpha_K^0(A) + \alpha_K^0(B))_{K\cdot} &= \alpha_K^0(\bigsqcup_{i\geq K}(\alpha_K^0(A) + \alpha_K^0(B))_{i\cdot}) \\ &= \alpha_K^0((\bigsqcup_{i\geq K} \alpha_K^0(A)_{i\cdot}) + (\bigsqcup_{i\geq K} \alpha_K^0(B)_{i\cdot})) \\ &= \alpha_K^0(\alpha_K^0(\bigsqcup_{i\geq K} A_{i\cdot}) + \alpha_K^0(\bigsqcup_{i\geq K} B_{i\cdot})) \\ &= \alpha_K^0((\bigsqcup_{i\geq K} A_{i\cdot}) + (\bigsqcup_{i\geq K} B_{i\cdot})) \\ &= \alpha_K^0(\bigsqcup_{i\geq K}(A + B)_{i\cdot}) \\ &= \alpha_K^0(A + B)_{K\cdot}\ .\end{aligned}$$

□

We also prove the corresponding theorem for quads. Note the inequality as opposed to the equality in the preceding proposition.

Theorem 4.42 *Let $Q = (s, A, t, u) \in Q_{mn}$ and $Q' = (s', A', t', u') \in Q_{mk}$. Then*

$$\alpha_K^0(Q + Q') \sqsubseteq \alpha_K^0(\alpha_K^0(Q) + \alpha_K^0(Q')).$$

Proof. We exploit that α_K^0 is the identity on vectors of length less than or equal to K.

$$\begin{aligned}&\alpha_K^0(Q + Q') \\ &= \alpha_K^0(s + s', A + A', t + t', u + u') \\ &= (\alpha_K^0(s + s'), \alpha_K^0(A + A'), \alpha_K^0(t + t'), u + u') \\ &\sqsubseteq (\alpha_K^0(\alpha_K^0(s) + \alpha_K^0(s')), \alpha_K^0(\alpha_K^0(A) + \alpha_K^0(A')), \alpha_K^0(t) + \alpha_K^0(t'), u + u') \\ &= (\alpha_K^0(\alpha_K^0(s) + \alpha_K^0(s')), \alpha_K^0(\alpha_K^0(A) + \alpha_K^0(A')), \alpha_K^0(\alpha_K^0(t) + \alpha_K^0(t')), u + u') \\ &= \alpha_K^0(\alpha_K^0(s) + \alpha_K^0(s'), \alpha_K^0(A) + \alpha_K^0(A'), \alpha_K^0(t) + \alpha_K^0(t'), u + u') \\ &= \alpha_K^0(\alpha_K^0(Q) + \alpha_K^0(Q')).\end{aligned}$$

□

4.8 Abstracting Sets of Quads

We approximate a set of quads with a $(K+1)\times(K+1)$ matrix of quads, called a K-*grid* in the following. We first adjoin a bottom element to the set of $m \times n$ quads over a **DAlg**-structure D to be able to represent an empty set of quads of that size.

Definition 4.43 Let $Q_{mn}(D)$ be the set of $m\times n$ quads over the **DAlg**-structure D, equipped with the quad ordering $\sqsubseteq$ inherited from D. We define

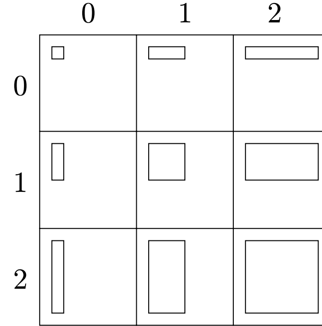
$$Q_{mn}^{\perp}(D) = Q_{mn}(D) \uplus \{\perp\},$$

and extend the ordering such that $\perp \sqsubseteq Q$ for any $Q \in Q_{mn}^{\perp}(D)$. The “dimension” of $\perp$ will be implied by context. □

Quad composition and concatenation are extended to be strict in $\perp$: $\perp Q = Q\perp = \perp$, and $\perp +\!\!\!+ Q = Q +\!\!\!+ \perp = \perp$.

Definition 4.44 A K-*grid* is a $(K + 1) \times (K + 1)$ matrix of quads, such that for $i, j \in \{0, 1, \ldots , K\}$, whenever G is a K-grid then $G_{ij} \in Q_{ij}^{\perp}$. □

We will often omit the K when the constant is obvious from the context and merely talk about grids. We extend the ordering of quads to grids in a componentwise manner. The following picture illustrates a 2-grid:



Definition 4.45 For two K-grids G and G' define

$$G \sqsubseteq G' \iff \forall i, j \leq K : G_{ij} \sqsubseteq G'_{ij}.$$

□

As we defined the operations of sequential composition and concatenation of quad languages, we can now define corresponding operations on grids that mimic these operations. We want sequential composition of grids to be a sound approximation of sequential composition of languages, and similarly for concatenation.

Definition 4.46 Let G and G' be K-grids. We define their sequential composition componentwise as:

$$(GG')_{ij} = \bigsqcup_{0\leq k\leq K} G_{ik}G'_{kj}.$$

□

Computing $(GG')_{ij}$ requires at most K least upper bounds and K multiplications of quads. We may write the cost of computing $(GG')_{ij}$ as:

$$\begin{aligned}\sum_{k=0}^{K}((i+1)(j+1)+(i+1)(k+1)(j+1)) &= \sum_{k=0}^{K}(k+2)(i+1)(j+1)\\ &\leq (i+1)(j+1)(K+2)^2,\end{aligned}$$

thus the total cost of computing GG' is:

$$(K+2)^2\sum_{i=0}^{K}\sum_{j=0}^{K}(i+1)(j+1) \leq (K+2)^2(K+1)^2(K+1)^2,$$

and the complexity of grid multiplication is therefore in $O(K^6)$.

The concatenation of two elements (quads) of a grid may be too large to fit into a grid of the same size, so a second application of the α_K^0 abstraction function may be necessary.

Definition 4.47 Let G and G' be K-grids. For $i \leq K$ and $j < K$ define:

$$(G \mathbin{+\!\!+} G')_{ij} = \bigsqcup\{G_{in} \mathbin{+\!\!+} G'_{im} \mid n+m=j\},$$

and finally define:

$$(G \mathbin{+\!\!+} G')_{iK} = \bigsqcup\{\alpha_K^0(G_{in} \mathbin{+\!\!+} G'_{im}) \mid n+m \geq K\}.$$

□

Computing $(G \mathbin{+\!\!+} G')_{ij}$ for $j < K$ requires at most K least upper bound- and quad-concatenation operations, that is, the cost is dominated by $2K(i+1)(j+1)$. For $j = K$ the second half of the definition takes over and the computation needs on the order of K^2 concatenations, applications of α_K^0, and least upper bounds, thus the cost is dominated by $3K^2(i+1)(m+n+1)$. The cost of concatenating two entire K-grids can be estimated as:

$$2K\sum_{i=0}^{K}\sum_{j=0}^{K-1}(i+1)(j+1) \leq 2K(K+1)^2K^2 \in O(K^5),$$

for the $j < K$ case, and for the remaining case:

$$3K^2\sum_{i=0}^{K}(i+1)(2K+1) \in O(K^5).$$

So the upper bound on grid concatenation is in $O(K^5)$. Computing the l.u.b. of two K-grids performs K^2 least upper bounds of quads at most of size $K \times K$, thus the complexity of this operation is in $O(K^4)$.

We can now define the abstraction of quad languages into grids.

Definition 4.48 Let L be a set of quads (a language) over a **DAlg**-structure D. Define for $i, j \leq K$:

$$\alpha_K(L)_{ij} = \bigsqcup\{\alpha_K^0(Q) \mid Q \in L \text{ and } \alpha_K^0(Q) \in Q_{ij}(D)\}.$$

□

In effect, for $i, j < K$, $\alpha_K(L)_{ij}$ is the approximation of all the quads in the set L of size $i \times j$. The quad $\alpha_K(L)_{iK}$ approximates all the quads in the set with i rows and at least K columns.

The following shorthand notation will prove useful in the forthcoming proofs, and gives a more elegant formulation of α_K.

Definition 4.49 Let L be a quad language. Define

$$L_{mn} = \{Q \in L \mid \alpha_K^0(Q) \in Q_{mn}(D)\}.$$

That is, L_{mn} is the subset of quads in L whose abstraction is a $m \times n$ quad. Note that this definition depends on the value of K. We leave out an explicit reference to K as its value will be clear from context. □

With the above definition, we can rephrase the definition of α_K as:

$$\alpha_K(L)_{ij} = \bigsqcup_{Q \in L_{ij}} \alpha_K^0(Q).$$

It is not hard to see that α_K is monotone with respect to subset inclusion: if $L \subseteq L'$ then $\alpha_K(L) \sqsubseteq \alpha_K(L')$. If we define γ_K as the pre-image of α_K:

$$\gamma_K(G) = \bigcup_{\alpha_K(L)=G} L$$

we then get a pair of adjoined functions between sets of quads ordered by subset inclusion and grids ordered by the $\sqsubseteq$ ordering:

$$\alpha_K(L) \sqsubseteq G \iff L \subseteq \gamma_K(G).$$

Now we are finally in a position to prove the soundness of our abstract operations, namely the inequalities:

$$\begin{aligned} \alpha_K(LL') &\sqsubseteq \alpha_K(L)\alpha_K(L'), \\ \alpha_K(L \mathbin{+\!\!\!+} L') &\sqsubseteq \alpha_K(L) \mathbin{+\!\!\!+} \alpha_K(L'), \\ \alpha_K(L \cup L') &\sqsubseteq \alpha_K(L) \sqcup \alpha_K(L'). \end{aligned}$$

We start out with the theorem for composition.

Theorem 4.50 *For quad languages L and L' we have the inequality:*

$$\alpha_K(LL') \sqsubseteq \alpha_K(L)\alpha_K(L').$$

Proof. For $i, j \leq K$ we compute, using monotonicity of quad composition and Theorem 4.37:

$$\begin{aligned} (\alpha_K(L)\alpha_K(L'))_{ij} &= \bigsqcup_{0 \leq k \leq K} \alpha_K(L)_{ik}\alpha_K(L')_{kj} \\ &= \bigsqcup_{0 \leq k \leq K} ((\bigsqcup_{Q \in L_{ik}} \alpha_K^0(Q))(\bigsqcup_{Q' \in L'_{kj}} \alpha_K^0(Q'))) \\ &\sqsupseteq \bigsqcup_{0 \leq k \leq K} \bigsqcup_{Q \in L_{ik}} \bigsqcup_{Q' \in L'_{kj}} \alpha_K^0(Q)\alpha_K^0(Q') \\ &\sqsupseteq \bigsqcup_{0 \leq k \leq K} \bigsqcup_{Q \in L_{ik}} \bigsqcup_{Q' \in L'_{kj}} \alpha_K^0(QQ'). \end{aligned} \tag{4.1}$$

To prove the connection to $\alpha_K(LL')$ we first compute:

$$\begin{aligned}(LL')_{ij} &= \{Q'' \in LL' \mid \alpha^0_K(Q'') \in Q_{ij}\} \\ &= \{Q'' \in \{QQ' \mid Q \in L \cap Q_{mn}, Q' \in L' \cap Q_{nk}\} \mid \alpha^0_K(Q'') \in Q_{ij}\} \\ &= \{QQ' \mid Q \in L \cap Q_{mn}, Q' \in L' \cap Q_{nk}, \alpha^0_K(QQ') \in Q_{ij}\}.\end{aligned}$$

We can then simplify:

$$\begin{aligned}\alpha_K(LL')_{ij} &= \bigsqcup_{Q'' \in (LL')_{ij}} \alpha^0_K(Q'') \\ &= \bigsqcup_{\{QQ' \mid Q \in L \cap Q_{mn}, Q' \in L' \cap Q_{nk}, \alpha^0_K(QQ') \in Q_{ij}\}} \alpha^0_K(QQ'). \qquad (4.2)\end{aligned}$$

Now we just need to show that the triple l.u.b. (4.1) ranges over a larger set of quads than the l.u.b. (4.2), that is: If $Q \in L \cap Q_{mn}, Q' \in L' \cap Q_{nk}$, and $\alpha^0_K(QQ') \in Q_{ij}$ then there is a $k' \in \{0, \ldots, K\}$ such that $Q \in L_{ik'}$ and $Q' \in L'_{k'j}$.

The consequent of that implication can also be phrased as: there is a $k' \in \{0, \ldots, K\}$ such that $Q \in L$, $\alpha^0_K(Q) \in Q_{ik'}$, $Q' \in L'$, and $\alpha^0_K(Q') \in Q_{k'j}$. That the implication holds with $k' = \min(K, n)$ is then easily seen from the definition of α^0_K. This completes the proof of the desired inequality. □

Theorem 4.51 *For quad languages L and L' we have the inequality:*

$$\alpha_K(L + L') \sqsubseteq \alpha_K(L) + \alpha_K(L').$$

Proof. There are two cases according to the definition of concatenation for grids. For $j < K$ we calculate, as in the previous proof, using monotonicity of concatenation and Theorem 4.42:

$$\begin{aligned}(\alpha_K(L) + \alpha_K(L'))_{ij} &= \bigsqcup_{n+k=j} ((\bigsqcup_{Q \in L_{in}} \alpha^0_K(Q)) + (\bigsqcup_{Q' \in L'_{ik}} \alpha^0_K(Q'))) \\ &\sqsupseteq \bigsqcup_{n+k=j} \bigsqcup_{Q \in L_{in}} \bigsqcup_{Q' \in L'_{ik}} \alpha^0_K(Q) + \alpha^0_K(Q') \\ &= \bigsqcup_{n+k=j} \bigsqcup_{Q \in L_{in}} \bigsqcup_{Q' \in L'_{ik}} \alpha^0_K(\alpha^0_K(Q) + \alpha^0_K(Q')) \\ &\sqsupseteq \bigsqcup_{n+k=j} \bigsqcup_{Q \in L_{in}} \bigsqcup_{Q' \in L'_{ik}} \alpha^0_K(Q + Q'). \qquad (4.3)\end{aligned}$$

As in the previous proof we compute (for any j):

$$\begin{aligned}(L + L')_{ij} &= \{Q'' \in L + L' \mid \alpha^0_K(Q'') \in Q_{ij}\} \\ &= \{Q'' \in \{Q + Q' \mid Q \in L \cap Q_{mn}, Q' \in L' \cap Q_{mk}\} \mid \alpha^0_K(Q'') \in Q_{ij}\} \\ &= \{Q + Q' \mid Q \in L \cap Q_{mn}, Q' \in L' \cap Q_{mk}, \alpha^0_K(Q + Q') \in Q_{ij}\},\end{aligned}$$

which allows us to simplify:

$$\begin{aligned}
&\alpha_K(L \mathbin{+\!\!\!+} L')_{ij} \\
&= \bigsqcup_{Q'' \in (L \mathbin{+\!\!\!+} L')_{ij}} \alpha^0_K(Q'') \\
&= \bigsqcup_{\{Q \mathbin{+\!\!\!+} Q' \ \mid\ Q \in L \cap \mathcal{Q}_{mn}, Q' \in L' \cap \mathcal{Q}_{mk}, \alpha^0_K(Q \mathbin{+\!\!\!+} Q') \in \mathcal{Q}_{ij}\}} \alpha^0_K(Q \mathbin{+\!\!\!+} Q'). \qquad (4.4)
\end{aligned}$$

To show the required inequality for $j < K$ we now just need to show the implication: If $Q \in L \cap \mathcal{Q}_{mn}$, $Q' \in L' \cap \mathcal{Q}_{mk}$, and $\alpha^0_K(Q \mathbin{+\!\!\!+} Q') \in \mathcal{Q}_{ij}$ then there exists n' and k' such that $n' + k' = j$, $Q \in L_{in'}$, and $Q' \in L'_{ik'}$. As $j < K$ we must have the equality $n + k = j$ (both Q and Q' has a small number of columns), so choosing $n' = n$ and $k' = k$ easily verifies the implication. Thus the l.u.b. in (4.3) is over a larger set than the l.u.b. in (4.4).

For the case $j = K$ we use Definition 4.47, monotonicity of concatenation, and Prop. 4.33 to compute:

$$\begin{aligned}
(\alpha_K(L) \mathbin{+\!\!\!+} \alpha_K(L'))_{iK} &= \bigsqcup_{n+m \geq K} \alpha^0_K(\alpha_K(L)_{in} \mathbin{+\!\!\!+} \alpha_K(L')_{im}) \\
&= \bigsqcup_{n+m \geq K} \alpha^0_K((\bigsqcup_{Q \in L_{in}} \alpha^0_K(Q)) \mathbin{+\!\!\!+} (\bigsqcup_{Q' \in L'_{im}} \alpha^0_K(Q'))) \\
&\sqsupseteq \bigsqcup_{n+m \geq K} \alpha^0_K(\bigsqcup_{Q \in L_{in}} \bigsqcup_{Q' \in L'_{im}} \alpha^0_K(Q) \mathbin{+\!\!\!+} \alpha^0_K(Q')) \\
&= \bigsqcup_{n+m \geq K} \bigsqcup_{Q \in L_{in}} \bigsqcup_{Q' \in L'_{im}} \alpha^0_K(\alpha^0_K(Q) \mathbin{+\!\!\!+} \alpha^0_K(Q')) \\
&\sqsupseteq \bigsqcup_{n+m \geq K} \bigsqcup_{Q \in L_{in}} \bigsqcup_{Q' \in L'_{im}} \alpha^0_K(Q \mathbin{+\!\!\!+} Q'). \qquad (4.5)
\end{aligned}$$

Again we need to show that the l.u.b. in (4.5) ranges over a larger set than the l.u.b. in (4.4). That is, if $Q \in L \cap \mathcal{Q}_{mn}$, $Q' \in L' \cap \mathcal{Q}_{mk}$, and $\alpha^0_K(Q \mathbin{+\!\!\!+} Q') \in \mathcal{Q}_{iK}$ then there exists n' and m' such that $n' + m' \geq K$, $Q \in L_{in'}$, and $Q' \in L'_{im'}$. As $\alpha^0_K(Q \mathbin{+\!\!\!+} Q') \in \mathcal{Q}_{iK}$ there must be n_0 and k_0 such that $n_0 + k_0 \geq K$, $Q \in L \cap \mathcal{Q}_{mn_0}$, and $Q' \in L' \cap \mathcal{Q}_{mk_0}$. Take $n' = n_0$ and $m' = k_0$ to verify the implication. □

We have already seen that α_K is a monotone function from quad languages ordered by subset inclusion to grids with the $\sqsubseteq$ ordering. This implies the inequality: $\alpha_K(L) \sqcup \alpha_K(L') \sqsubseteq \alpha_K(L \cup L')$. The following theorem shows the reverse inequality, stating that α_K is a join homomorphism.

Theorem 4.52 *Let L and L' be quad languages. Then*

$$\alpha_K(L \cup L') = \alpha_K(L) \sqcup \alpha_K(L').$$

Proof. It is not hard to see that $(L \cup L')_{ij} = L_{ij} \cup L'_{ij}$. Using this fact we can compute:

$$(\alpha_K(L) \sqcup \alpha_K(L'))_{ij} \quad = \quad \alpha_K(L)_{ij} \sqcup \alpha_K(L')_{ij}$$

$$
\begin{aligned}
&= (\bigsqcup_{Q \in L_{ij}} \alpha_K^0(Q)) \sqcup (\bigsqcup_{Q' \in L'_{ij}} \alpha_K^0(Q')) \\
&= \bigsqcup_{Q \in (L \cup L')_{ij}} \alpha_K^0(Q) \\
&= \alpha_K(L \cup L')_{ij}.
\end{aligned}
$$

□

If we define a *vector-grid* V by analogy with grids as a vector of vectors such that the i'th entry V_i of the vector-grid is a vector of length i, then we can define the image of a vector-grid through a grid as follows.

Definition 4.53 Let V be a K vector-grid and G be a K-grid then $\mathrm{img}(V, G)$ is a K vector-grid with:

$$\mathrm{img}(V, G)_j = \bigsqcup_{0 \le i \le K} \mathrm{img}(V_i, G_{ij}).$$

□

We may also extend the definition of α_K to map sets of vectors to vector-grids in the obvious manner.

By a computation similar to the proof of the corresponding theorem for quad composition, using Corollary 4.38, we obtain the following:

Proposition 4.54 *Let V be a set of vectors, and L a quad language. Then*

$$\alpha_K(\mathrm{img}(V, L)) \sqsubseteq \mathrm{img}(\alpha_K(V), \alpha_K(L)).$$

A similar definition and proposition for pre-images can derived by analogous means, but will not be needed in the following.

4.9 Soundness

This section formulates the general kind of soundness property that one would want for an analysis based on the dependence algebra technology developed here.

Suppose p is a program mapping stores to stores: $p : (S \to V) \to (S \to V)$, where S is a set of slots, and V is the set of values. Let α be some abstraction function that abstracts values into properties of interest as in abstract interpretation. If the analysis derives a dependence quad $[\![p]\!]$ from the program in some fashion, then the soundness of the analysis can be formulated as the following property:

Property 4.55 *Let s and s' be stores. If $p(s) = s'$ then*

$$
\begin{aligned}
\alpha(s') &\sqsubseteq \mathrm{img}(\alpha(s), [\![p]\!]), \\
\alpha(s) &\sqsubseteq \mathrm{pre}([\![p]\!], \alpha(s')).
\end{aligned}
$$

The first inequation is just a reformulation of the usual soundness property in abstract interpretation, whereas the second inequation is dual of this, corresponding to the fact that quads can be "run" backwards, and thereby provide backwards information flow as well as forward information flow.

More concrete formulations of soundness are given in the subsequent chapters where concrete analyses using the quad framework are presented.

4.10 Constructing a DAlg-structure

This section describes several ways of constructing **DAlg**-structures from simpler structures, in particular from distributive lattices. Recall that a distributive lattice $(L, \leq, \vee, \wedge)$ is a lattice where $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$ for all elements $a, b, c \in L$.

The simplest **DAlg**-structure is the one-point set with all operations being trivial. Another simple construction of a **DAlg**-structure from a distributive lattice $(L, \leq, \vee, \wedge)$ is to use the l.u.b. of the lattice as both $+$ and $\sqcup$, and use the g.l.b. for sequential composition. It is not hard to see that all the axioms of a **DAlg**-structure are fulfilled with this construction.

A more elaborate, but important, construction of a **DAlg**-structure from a distributive lattice $(L, \leq, \vee, \wedge)$ is the one developed in the rest of this section. It will be used in the soft type inference for actions in Chapter 6. The same construction can be used to construct a **DAlg**-structure for a strictness analysis.

The main idea of the construction is to distinguish between *may-* and *must-*dependences. Consider the program:

```
a := b;
if condition then
  x := y;
else
  x := z;
end if
```

after executing this piece of code we know that x *may* depend on either y or z, as the condition is dynamic. In contrast we know that a *must* depend on b. This distinction is formalized in the following.

We start from a distributive lattice (think of a sub-typing lattice) $(L, \leq, \vee, \wedge)$. We take two copies of this lattice, and overline the elements of one copy. The overlined elements correspond to must-dependences and the non-overlined elements correspond to may-dependences. We also add a new element, indep, for a “known” independence between two slots. We write $\uplus$ for disjoint set union.

Definition 4.56 Let L be the carrier set of a distributive lattice. Define

$$\overline{L}_{\mathsf{indep}} = L \uplus \{\overline{a} \mid a \in L\} \uplus \{\mathsf{indep}\}.$$

□

On the carrier set $\overline{L}_{\mathsf{indep}}$ we define the three operations of l.u.b., sum, and product that will make it into a **DAlg**-structure.

Adding two dependences represents the combination of parallel execution paths where both paths are taken. Adding a must-dependence to a may-dependence represents such a combination where the may-dependence path may not be taken. To ensure a sound representation, the result of the sum has to become a may-dependence. The sequential composition of a may- and a must-dependence also has to become a may-dependence for similar reasons. The join of two dependences is defined much as the sum, except for the treatment of independence. Independence should be neutral with respect to sum, but as join corresponds to

the combination of parallel execution paths where only one of the paths are taken, joining a must-dependence with independence, should give a may-dependence, as the must-dependence path might not be taken.

Definition 4.57 Let $(L, \leq, \vee, \wedge)$ be a distributive lattice. Construct the set $\overline{L}_{\mathsf{indep}}$ as above. We define the three operations: $+$, sequential composition, and $\sqcup$ as follows. For $a, b \in L$ define:

$$\begin{array}{rclcrcl} a + b & = & a \vee b & \qquad & \overline{a} + b & = & a \vee b \\ a + \overline{b} & = & a \vee b & & \overline{a} + \overline{b} & = & \overline{a \vee b} \end{array}$$

$$\begin{array}{rclcrcl} ab & = & a \wedge b & \qquad & \overline{a}b & = & a \wedge b \\ a\overline{b} & = & a \wedge b & & \overline{a}\overline{b} & = & \overline{a \wedge b}. \end{array}$$

For the indep case: for any $x \in \overline{L}_{\mathsf{indep}}$:

$$\mathsf{indep} + x = x = x + \mathsf{indep}, \text{ and } \mathsf{indep}\ x = x\ \mathsf{indep} = \mathsf{indep},$$

This makes indep into the zero, $\mathbf{0}$, of the structure. The unit, $\mathbf{1}$, of $\overline{L}_{\mathsf{indep}}$ is the overlined copy of the top of the original lattice, L. We define $\sqcup$ to be commutative, and for all $a, b \in L$ satisfy:

$$\begin{array}{lclcrcl} \mathsf{indep} \sqcup \mathsf{indep} & = & \mathsf{indep} & \qquad & a \sqcup b & = & a \vee b \\ a \sqcup \mathsf{indep} & = & a & & a \sqcup \overline{b} & = & a \vee b \\ \overline{a} \sqcup \mathsf{indep} & = & a & & \overline{a} \sqcup \overline{b} & = & \overline{a \vee b}. \end{array}$$

□

The above construction makes the $\sqsubseteq$ ordering of $\overline{L}_{\mathsf{indep}}$ into a join semi-lattice *without a bottom element*. Also note that none of the new operations are l.u.b. or g.l.b. with respect to the $\leq$ ordering.

In the following, as well as in the above definition, we use the convention that variables ranging over $\overline{L}_{\mathsf{indep}}$ are named $x, y, z, \ldots$, whereas variables ranging over L are named $a, b, c, \ldots$

The lack of a bottom element means that monotone endo-functions on the $\overline{L}_{\mathsf{indep}}$ structure do not necessarily have a *least* fixed point. This also means that we shall use the method described in Section 4.2.1 to compute approximations of fixed points for this **DAlg**-structure.

The alternative of adding an artificial bottom element to the structure was also considered, but extending the other operations to operate sensibly on the extra element proved to be cumbersome. The lack of a proper semantic explanation of the artificial bottom also was a reason for not adding such an element.

There are two minimal elements of the $\overline{L}_{\mathsf{indep}}$ structure with respect to the $\sqsubseteq$ ordering: the overlined copy of the bottom of the original lattice L, and the new element: indep. The indep element corresponds to not running the code at all as that would leave the outputs independent of the inputs. So starting iteration from indep does not make sense in a must/may analysis, as there would be no must-dependences if we always had to take the possibility of not running the function at all into consideration.

In the following we investigate the structure of the $\overline{L}_{\mathsf{indep}}$ construction above and argue that it satisfies the axioms of a **DAlg**-structure.

The $\sqcup$ operation is easily seen to be commutative and idempotent since $\vee$ is. By using the associativity of $\vee$ and examining each of the 27 cases of x, y, and z being of the forms $a, \overline{a}$, and indep, one can verify that $\sqcup$ is associative.

We next examine the ordering introduced the the $\sqcup$ operator. It is reflexive and transitive by the idempotency and associativity of $\sqcup$, respectively. Let $a \in L$ then

$$\mathsf{indep} \sqcup a = a \iff \mathsf{indep} \sqsubseteq a,$$

that is, indep is below all bare elements of $\overline{L}_{\mathsf{indep}}$. However, it is not below overlined elements:

$$\mathsf{indep} \sqcup \overline{a} \neq \overline{a} \iff \mathsf{indep} \not\sqsubseteq \overline{a}.$$

The $\sqsubseteq$ ordering is related to the $\leq$ ordering of the lattice L via the following: for $a, b \in L$:

$$a \leq b \iff a \vee b = b \iff a \sqcup b = b \iff a \sqsubseteq b,$$

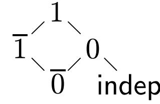
and

$$a \leq b \iff a \vee b = b \iff \overline{a \vee b} = \overline{b} \iff \overline{a} \sqcup \overline{b} = \overline{b} \iff \overline{a} \sqsubseteq \overline{b}.$$

The relation between overlined and non-overlined elements is determined by:

$$\overline{a} \sqsubseteq b \iff b = \overline{a} \sqcup b = a \vee b \iff a \leq b.$$

As an example consider the two point lattice $S = (\{0,1\}, \leq, \vee, \wedge)$ with $0 \leq 1$. The Hasse diagram of the $\sqsubseteq$ ordering of $\overline{S}_{\mathsf{indep}}$ is pictured below:



The commutativity of $+$ follows from the commutativity of $\wedge$, and idempotency of $+$ follows likewise. Associativity of $+$ is proved as for $\sqcup$ by examining each of the 27 cases and using associativity of $\wedge$. The neutral element for $+$ ($\mathbf{0}$) is indep. We formulate the monotonicity of $+$ as the following lemma.

Lemma 4.58 *Let $x, y, z \in \overline{L}_{\mathsf{indep}}$. Then*

$$x \sqsubseteq y \Rightarrow x + z \sqsubseteq y + z.$$

Proof. Let $a, b, c \in L$. There are 4 cases, and for each case z may be of one of the forms indep, c, or $\overline{c}$. For $z = \mathsf{indep}$ the result is trivial because indep is neutral for $+$.

1. $x = a \sqsubseteq b = y$: if $z = c$ we have $x + z = a + c = a \vee c \leq b \vee c = b + c = y + z$. If $z = \overline{c}$ we have $x + z = a + \overline{c} = a \vee c \leq b \vee c = b + \overline{c} = y + z$. In both cases we use the above characterization of $\sqsubseteq$ to get the desired inequality.

2. $x = \mathsf{indep} \sqsubseteq b = y$: if $z = c$ we have $x + z = \mathsf{indep} + c = c \leq b \vee c = y + z$. If $z = \overline{c}$ we have $x + z = \mathsf{indep} + \overline{c} = \overline{c} \sqsubseteq c \leq b \vee c = y + z$.

3. $x = \overline{a} \sqsubseteq b = y$: if $z = c$ we have $x + z = \overline{a} + c = a \vee c \leq b \vee c = y + z$. If $z = \overline{c}$ then $x + z = \overline{a} + \overline{c} = \overline{a \vee c} \sqsubseteq a \vee c \leq b \vee c = z + y$.

4. $x = \overline{a} \sqsubseteq \overline{b} = y$: if $z = c$ we have $x + z = \overline{a} + c = a \vee c \leq b \vee c = \overline{b} + c = y + z$. If $z = \overline{c}$ then $x + z = \overline{a} + \overline{c} = \overline{a \vee c} \sqsubseteq \overline{b \vee c} = \overline{b} + \overline{c} = y + z$.

□

The previous arguments have ensured that the **DAlg** axioms for $\sqcup$ and $+$ are satisfied. Next we investigate the sequential composition operation. The unit $\mathbf{1}$ of the **DAlg**-structure was defined as the overlined copy of the top of the underlying lattice, so it is not hard to see that the (unit) axiom is satisfied. In the two point lattice example above, we get $\mathbf{1} = \overline{1}$. Likewise, the (annihilator) axiom is satisfied by construction: $\mathbf{0} = \mathsf{indep}$.

We formulate the distributivity of $\cdot$ over $+$ as a lemma.

Lemma 4.59 *Let $x, y, z \in \overline{L}_{\mathsf{indep}}$. Then*

$$x(y + z) = xy + xz.$$

Proof. If $x = \mathsf{indep}$ both sides are indep. If $y = \mathsf{indep}$ then $x(\mathsf{indep} + z) = xz = \mathsf{indep} + xz = x \cdot \mathsf{indep} + xz$, and likewise for $z = \mathsf{indep}$. There are therefore 8 cases to consider: each of x, y, z may be overlined or not. Using the distributivity of L, that: $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$ it is trivial to verify each case. □

It is not hard to see that the definition of $\cdot$ is commutative in this particular case, so distributivity from the other side follows.

It now only remains to verify the monotonicity of the sequential composition. The proof follows the same idea as the monotonicity of $+$.

Lemma 4.60 *Let $x, y, z \in \overline{L}_{\mathsf{indep}}$. Then*

$$x \sqsubseteq y \Rightarrow xz \sqsubseteq yz.$$

Proof. Let $a, b, c \in L$. There are 4 cases, and for each case z may be of one of the forms indep, c, or $\overline{c}$. For $z = \mathsf{indep}$ the result is trivial because indep is an annihilator for sequential composition.

1. $x = a \sqsubseteq b = y$: if $z = c$ we have $xz = ac = a \wedge c \leq b \wedge c = bc = yz$. If $z = \overline{c}$ we have $xz = a\overline{c} = a \wedge c \leq b \wedge c = b\overline{c} = yz$. In both cases we use the above characterization of $\sqsubseteq$ to get the desired inequality.

2. $x = \mathsf{indep} \sqsubseteq b = y$: if $z = c$ we have $xz = \mathsf{indep} \leq b \wedge c = yz$. If $z = \overline{c}$ we have $xz = \mathsf{indep} \leq b \wedge c = yz$.

3. $x = \overline{a} \sqsubseteq b = y$: if $z = c$ we have $xz = \overline{a}c = a \wedge c \leq b \wedge c = yz$. If $z = \overline{c}$ then $xz = \overline{a}\overline{c} = \overline{a \wedge c} \sqsubseteq a \wedge c \leq b \wedge c = zy$.

4. $x = \overline{a} \sqsubseteq \overline{b} = y$: if $z = c$ we have $xz = \overline{a}c = a \wedge c \leq b \wedge c = \overline{b}c = yz$. If $z = \overline{c}$ then $xz = \overline{a}\overline{c} = \overline{a \wedge c} \sqsubseteq \overline{b \wedge c} = \overline{b}\overline{c} = yz$.

□

We summarize the preceding arguments into the following theorem.

Theorem 4.61 *Let L be a distributive lattice. The algebraic structure $\overline{L}_{\mathsf{indep}}$ constructed in Definition 4.57 is a* **DAlg***-structure.*

4.11 Reverse Images

In soft type inference we only want to require an input argument to be of a certain type if we are sure it is needed to be of that type. If the argument needs to be an integer in only one of the branches of a conditional we will not require it to be an integer.

The foundation for the must/may distinction among dependences was established in the previous section, but the pre-image construction given earlier does not take advantage of this distinction, as it treats all dependences equally.

This section defines the notion of a *reverse image* which discards information about may-dependences, and computes more accurate information about necessary input types using the must-dependences.

The soundness property that we want from an analysis built on these instruments is that: if a program executes successfully then the reverse image of its output through the quad that models the program is a safe approximation of the actual input that the program received from the start.

Definition 4.62 Let L be a distributive lattice with top $\top$, and let $a \in L$. Define the function $\Psi : \overline{L}_{\mathsf{indep}} \to L$ as:

$$\begin{aligned} \Psi(\mathsf{indep}) &= \top \\ \Psi(a) &= \top \\ \Psi(\overline{a}) &= a. \end{aligned}$$

$\square$

The intuition that one should have about the Ψ function is that it "forgets" may-dependences and independence by converting them to top. This allows us to retain only the must-dependences of a quad, which are then used to compute the reverse image of an output vector.

To establish a number of inequalities we compute the following table: Let $x, y \in \overline{L}_{\mathsf{indep}}$, and $a, b \in L$.

x	y	$\Psi(x)$	$\Psi(y)$	$x \sqcup y$	$x + y$	xy	$\Psi(x \sqcup y)$	$\Psi(x + y)$	$\Psi(xy)$
indep	indep	$\top$	$\top$	indep	indep	indep	$\top$	$\top$	$\top$
a	indep	$\top$	$\top$	a	a	indep	$\top$	$\top$	$\top$
$\overline{a}$	indep	a	$\top$	a	$\overline{a}$	indep	$\top$	a	$\top$
indep	b	$\top$	$\top$	b	b	indep	$\top$	$\top$	$\top$
a	b	$\top$	$\top$	$a \vee b$	$a \vee b$	$a \wedge b$	$\top$	$\top$	$\top$
$\overline{a}$	b	a	$\top$	$a \vee b$	$a \vee b$	$a \wedge b$	$\top$	$\top$	$\top$
indep	$\overline{b}$	$\top$	b	b	$\overline{b}$	indep	$\top$	b	$\top$
a	$\overline{b}$	$\top$	b	$a \vee b$	$a \vee b$	$a \wedge b$	$\top$	$\top$	$\top$
$\overline{a}$	$\overline{b}$	a	b	$\overline{a \vee b}$	$\overline{a \vee b}$	$\overline{a \wedge b}$	$a \vee b$	$a \vee b$	$a \wedge b$

From the above table we can deduce the following inequalities relating algebraic operations on the $\overline{L}_{\mathsf{indep}}$ structure to the lattice structure of L via the Ψ function.

Proposition 4.63 *Let* $x, y \in \overline{L}_{\mathsf{indep}}$. *Then*

$$\Psi(x) \wedge \Psi(y) \leq \Psi(xy),$$

$$\Psi(x) \wedge \Psi(y) \leq \Psi(x + y) \leq \Psi(x) \vee \Psi(y).$$

We can also see from the table that Ψ is a join homomorphism and thereby monotone.

Proposition 4.64 *Let $x, y \in \overline{L}_{\mathsf{indep}}$. Then*

$$\Psi(x \sqcup y) = \Psi(x) \vee \Psi(y).$$

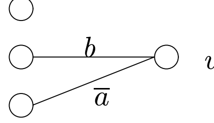
We define the reverse image of a vector v over L through a matrix as follows. The input required to get the output v must satisfy the conjunction of all the requirements on all the must-paths from input to output. Alluding to the low-pass filter analogy, the input must be of a sufficiently low frequency to pass through all the filters on the way from input to output. We write $\prod_{1 \leq i \leq n} x_i$ for the n-vector $(x_1, x_2, \ldots, x_n)$.

Definition 4.65 Define the *reverse image* of an n-vector v over L through a matrix $A \in M_{mn}(\overline{L}_{\mathsf{indep}})$ as:

$$\text{rev}(A, v) = \prod_i (\bigwedge_k \Psi(A_{ik} \overline{v_k})).$$

□

Consider the matrix pictured below, with three input slots and one output slot. The output is connected to the second input slot via a may-dependence b, and to the third input slot via a must dependence $\overline{a}$. The output is independent of the first input slot.



Suppose v is the type of the output, and we want the reverse image of v under the matrix. With the above definition of rev() we get $(\top, \top, v \wedge a) = \text{rev}(A, v)$.

For a quad over $\overline{L}_{\mathsf{indep}}$ we define the reverse image as follows. Here the input must satisfy the requirements of the matrix, as well as the requirements of the sink vector. We extend Ψ componentwise to vectors over $\overline{L}_{\mathsf{indep}}$.

Definition 4.66 Let $Q = (s, A, t, u) \in Q_{mn}(\overline{L}_{\mathsf{indep}})$, and let v an n-vector over L then define:

$$\text{rev}(Q, v) = \text{rev}(A, v) \wedge \Psi(t).$$

□

Observe that the definition of rev consists entirely of monotone functions (overlining is clearly monotone), so rev is monotone in both its arguments, stated formally below.

Proposition 4.67 *Let $Q, Q' \in Q_{mn}(\overline{L}_{\mathsf{indep}})$, and let v, v' be n-vectors over L. Then*

$$Q \sqsubseteq Q' \textit{ and } v \leq v' \Rightarrow \text{rev}(Q, v) \leq \text{rev}(Q', v').$$

The rev function is additive in its second argument.

Lemma 4.68 *Let $Q \in Q_{mn}(\overline{L}_{\mathsf{indep}})$, and let v, w be n-vectors over L. Then*

$$\text{rev}(Q, v \wedge w) = \text{rev}(Q, v) \wedge \text{rev}(Q, w).$$

Proof. By monotonicity we get the inequality:

$$\mathrm{rev}(Q, v \wedge w) \leq \mathrm{rev}(Q, v) \wedge \mathrm{rev}(Q, w).$$

The converse inequality follows by using idempotency and commutativity of the product in this particular **DAlg**-structure plus Prop. 4.63: Let $Q = (s, A, t, u)$, and compute:

$$\begin{aligned}
\mathrm{rev}(Q, v \wedge w)_i &= \bigwedge_k \Psi(A_{ik}\overline{(v \wedge w)_k}) \wedge \Psi(t_i) \\
&= \bigwedge_k \Psi(A_{ik}\overline{v_k w_k}) \wedge \Psi(t_i) \\
&= \bigwedge_k \Psi(A_{ik}A_{ik}\overline{v_k w_k}) \wedge \Psi(t_i) \\
&= \bigwedge_k \Psi(A_{ik}\overline{v_k}A_{ik}\overline{w_k}) \wedge \Psi(t_i) \\
&\geq \bigwedge_k \Psi(A_{ik}\overline{v_k}) \wedge \Psi(A_{ik}\overline{w_k}) \wedge \Psi(t_i) \\
&= \mathrm{rev}(Q, v)_i \wedge \mathrm{rev}(Q, w)_i \ .
\end{aligned}$$

□

We want to prove that taking the reverse image under the composition of two quads is less precise than taking the reverse image under each quad separately. First we need some intermediate results. The first lemma characterizes a part of the definition of rev.

Lemma 4.69 *Let $a, b \in L$. Then:*

$$\begin{aligned}
\Psi(\mathsf{indep}\ \overline{a}) &= \top \\
\Psi(b\overline{a}) &= \top \\
\Psi(\overline{b}\overline{a}) &= b \wedge a.
\end{aligned}$$

Proof. Simple computation. □

Lemma 4.70 *Let $x \in \overline{L}_{\mathsf{indep}}$, and $y \in L \cup \{\mathsf{indep}\}$. Then*

$$\Psi(x) \leq \Psi(x + y).$$

Proof. See the table on page 91. □

The next lemma relates rev to matrix multiplication and Ψ.

Lemma 4.71 *Let $A \in M_{mn}(\overline{L}_{\mathsf{indep}})$, and let v be an n vector over $\overline{L}_{\mathsf{indep}}$. Then*

$$\mathrm{rev}(A, \Psi(v)) \leq \Psi(Av).$$

Proof. Let $M = \{k \mid A_{ik} = \overline{b_k}\}$, that is, the set of column indices such that A_{ik} is overlined. Using first the definition of rev we compute:

$$\begin{aligned}
\mathrm{rev}(A, \Psi(v))_i &= \bigwedge_k \Psi(A_{ik}\overline{\Psi(v_k)}) \\
&= \bigwedge_{m\in M} b_m \wedge \Psi(v_m) \text{ (Lemma 4.69)} \\
&= \bigwedge_{m\in M} \Psi(\overline{b_m}) \wedge \Psi(v_m) \\
&\leq \bigwedge_{m\in M} \Psi(\overline{b_m}v_m) \\
&\leq \Psi(\sum_{m\in M} A_{im}v_m) \\
&\leq \Psi(\sum_k A_{ik}v_k) \text{ (Lemma 4.70)} \\
&= \Psi(Av)_i
\end{aligned}$$

□

The following proposition shows the desired result for matrices over $\overline{L}_{\mathsf{indep}}$.

Proposition 4.72 *Let $A \in M_{mn}(\overline{L}_{\mathsf{indep}})$, $A' \in M_{nl}(\overline{L}_{\mathsf{indep}})$, and let v be an l-vector over L. Then*

$$\mathrm{rev}(A, \mathrm{rev}(A', v)) \leq \mathrm{rev}(AA', v).$$

Proof. By definition we have:

$$\mathrm{rev}(A, \mathrm{rev}(A', v)) = \prod_i \bigwedge_j \Psi(A_{ij}\overline{\bigwedge_k \Psi(A'_{jk}\overline{v_k})}).$$

and

$$\mathrm{rev}(AA', v) = \prod_i \bigwedge_k \Psi((\sum_j A_{ij}A'_{jk})\overline{v_k}).$$

As the $\leq$ ordering is defined componentwise we need to show:

$$\bigwedge_j \Psi(A_{ij}\overline{\bigwedge_k \Psi(A'_{jk}\overline{v_k})}) \leq \bigwedge_k \Psi((\sum_j A_{ij}A'_{jk})\overline{v_k}) \tag{4.6}$$

for any i. If the right hand side is $\top$ there is nothing to prove. So suppose

$$\bigwedge_k \Psi((\sum_j A_{ij}A'_{jk})\overline{v_k}) < \top.$$

Then there is a non-empty set V of k's such that

$$(\sum_j A_{ij}A'_{jk})\overline{v_k} = \overline{a_k}$$

for some $a_k \in L$. By the definition of sum and product in this particular kind of **DAlg**-structure this in turn means that there are $b_k \in L$ such that for all $i \leq m$, and $k \in V$:

$$\sum_j A_{ij}A'_{jk} = \overline{b_k},$$

and consequently the existence of a non-empty set P of p's such that $c_{pk}, d_{pk} \in L$ and $A_{ip} = \overline{c_{pk}}$ and $A'_{pk} = \overline{d_{pk}}$. If there was a j such that $A_{ij}A'_{jk}$ is neither overlined nor indep then by the definition of $+$ the sum couldn't be overlined. With this we can now write:

$$\overline{b_k} = \sum_j A_{ij}A'_{jk} = \sum_{p\in P} A_{ip}A'_{pk}.$$

$$(\sum_{p\in P} A_{ip}A'_{pk})\overline{v_k} = \overline{(\bigvee_{p\in P} c_{pk} \wedge d_{pk}) \wedge v_k},$$

and in turn:

$$\Psi((\sum_j A_{ij}A'_{jk})\overline{v_k}) \quad = \quad (\bigvee_{p\in P} c_{pk} \wedge d_{pk}) \wedge v_k. \tag{4.7}$$

For all $k \in V$ and all $p \in P$ we have: $A'_{pk}\overline{v_k} = \overline{d_{pk} \wedge v_k}$, and in turn:

$$\bigwedge_{q\leq l} \Psi(A'_{pq}\overline{v_q}) \leq d_{pk} \wedge v_k,$$

as we take the greatest lower bound over a larger set. By monotonicity we then get for $p \in P$:

$$A_{ip}\overline{\bigwedge_q \Psi(A'_{pq}\overline{v_q})} \sqsubseteq \overline{c_{pk}}\,\overline{(d_{pk} \wedge v_k)}$$

this means that we get:

$$\Psi(A_{ip}\overline{\bigwedge_q \Psi(A'_{pq}\overline{v_q})}) \leq c_{pk} \wedge d_{pk} \wedge v_k$$

and

$$\bigwedge_j \Psi(A_{ij}\overline{\bigwedge_q \Psi(A'_{jq}\overline{v_q})}) \leq \bigwedge_{p\in P} \Psi(A_{ip}\overline{\bigwedge_q \Psi(A'_{pq}\overline{v_q})}),$$

therefore:

$$\bigwedge_j \Psi(A_{ij}\overline{\bigwedge_q \Psi(A'_{jq}\overline{v_q})}) \quad \leq \quad \bigwedge_{p\in P} c_{pk} \wedge d_{pk} \wedge v_k. \tag{4.8}$$

By comparing equations (4.7) and (4.8) and observing that:

$$\bigwedge_{p\in P} c_{pk} \wedge d_{pk} \wedge v_k \leq \bigvee_{p\in P} c_{pk} \wedge d_{pk} \wedge v_k,$$

for all $k \in V$, we get (4.6) as desired. □

Exploiting the two previous statements we can now prove that taking the reverse image under two quads separately is more precise than taking the reverse image under the composition of the two quads. Conversely, taking the reverse image of a vector under the composition of two quads is a safe approximation of the reverse image under the two quads separately. Like Theorem 4.16 on page 70 this will prove important in the soundness proof in Chapter 6.

Theorem 4.73 *Let $Q = (s, A, t, u) \in Q_{mn}(\overline{L}_{\mathsf{indep}})$, $Q' = (s', A', t', u') \in Q_{nl}(\overline{L}_{\mathsf{indep}})$, and let v be an l-vector over L. Then*

$$\mathrm{rev}(Q, \mathrm{rev}(Q', v)) \le \mathrm{rev}(QQ', v).$$

Proof. Using Lemma 4.68, Prop. 4.72 and Lemma 4.71 we compute:

$$\begin{aligned}
\mathrm{rev}(Q, \mathrm{rev}(Q', v)) &= \mathrm{rev}(A, \mathrm{rev}(Q, v)) \wedge \Psi(t) \\
&= \mathrm{rev}(A, \mathrm{rev}(A', v) \wedge \Psi(t')) \wedge \Psi(t) \\
&= \mathrm{rev}(A, \mathrm{rev}(A', v)) \wedge \mathrm{rev}(A, \Psi(t')) \wedge \Psi(t) \\
&\le \mathrm{rev}(AA', v) \wedge \Psi(At') \wedge \Psi(t) \\
&\le \mathrm{rev}(AA', v) \wedge \Psi(At' + t) \\
&= \mathrm{rev}(QQ', v).
\end{aligned}$$

□

The reason for calling the rev-function a more precise pre-image is embedded in the following proposition.

Proposition 4.74 *Let $Q = (s, A, t, u) \in Q_{mn}(\overline{L}_{\mathsf{indep}})$, and let v be an n-vector over $\overline{L}_{\mathsf{indep}}$. Then*

$$\mathrm{rev}(Q, \Psi(v)) \le \Psi(\mathrm{pre}(Q, v)).$$

Proof.

$$\begin{aligned}
\mathrm{rev}(Q, \Psi(v)) &= \mathrm{rev}(A, \Psi(v)) \wedge \Psi(t) \\
&\le \Psi(Av) \wedge \Psi(t) \\
&\le \Psi(Av + t) \\
&= \Psi(\mathrm{pre}(Q, v)).
\end{aligned}$$

□

The next lemma follows directly from the definition of $+$ and sequential composition on $\overline{L}_{\mathsf{indep}}$.

Lemma 4.75 *For $a, b \in L$ the following inequalities hold:*

$$\begin{aligned}
a + b &= a \vee b \ge a \wedge b, \\
ab &= a \wedge b.
\end{aligned}$$

The connection between rev and concatenation is given in the next theorem.

Proposition 4.76 *Let $A \in M_{mn}(\overline{L}_{\mathsf{indep}})$, $B \in M_{ml}(\overline{L}_{\mathsf{indep}})$, and let v be an n-vector over L, and w be an l-vector over L. Then*

$$\mathrm{rev}(A +\!\!+ B, v +\!\!+ w) = \mathrm{rev}(A, v) \wedge \mathrm{rev}(B, w).$$

Proof.

$$\begin{aligned}
\mathrm{rev}(A +\!\!+ B, v +\!\!+ w)_i &= \bigwedge_k \Psi((A +\!\!+ B)_{ik}\overline{(v +\!\!+ w)_k}) \\
&= (\bigwedge_{k \leq n} \Psi(A_{ik}\overline{v_k})) \wedge (\bigwedge_{n<k\leq n+l} \Psi(B_{i(k-n)}\overline{w_{k-n}})) \\
&= \mathrm{rev}(A, v)_i \wedge \mathrm{rev}(B, w)_i \ .
\end{aligned}$$

□

Theorem 4.77 *Let $Q = (s, A, t, u) \in Q_{mn}(\overline{L}_{\mathsf{indep}})$, $Q' = (s', A', t', u') \in Q_{ml}(\overline{L}_{\mathsf{indep}})$, let v be an n-vector over L, and let w be an l-vector over L. Then*

$$\mathrm{rev}(Q, v) \wedge \mathrm{rev}(Q', w) \leq \mathrm{rev}(Q +\!\!+ Q', v +\!\!+ w).$$

Proof.

$$\begin{aligned}
\mathrm{rev}(Q +\!\!+ Q', v +\!\!+ w) &= \mathrm{rev}((s +\!\!+ s', A +\!\!+ A', t + t', u + u'), v +\!\!+ w) \\
&= \mathrm{rev}(A +\!\!+ A', v +\!\!+ w) \wedge \Psi(t + t') \\
&= \mathrm{rev}(A, v) \wedge \mathrm{rev}(A', w) \wedge \Psi(t + t') \\
&\geq \mathrm{rev}(A, v) \wedge \mathrm{rev}(A', w) \wedge \Psi(t) \wedge \Psi(t') \\
&= \mathrm{rev}(Q, v) \wedge \mathrm{rev}(Q', w).
\end{aligned}$$

□

4.12 Discussion

As most other program analysis methods, the dependence algebra method can be placed in the framework of abstraction interpretation [CC77, CC79b]. Leaving out many formal details, that setup is as follows. Given a concrete semantics $\tau : P \to (A \to A)$, where P is the set of program terms, and A the semantic domain (integers, stores, predicates, ...), one defines a family $\overline{A}$ of subsets of A that characterizes the information of interest. From this family of subsets one can derive a "concretization" function $\gamma : L \to \overline{A}$, where L is a complete lattice representation of $\overline{A}$. The function γ has an adjoined function $\alpha : 2^A \to L$, the so-called abstraction function. From the given concrete semantics and the choice of $\overline{A}$, a best (most accurate) approximate semantic (abstract interpretation) function $t : P \to (L \to L)$ can be derived as $t(P)(a) = \alpha(\tau'(P)(\gamma(a)))$. Here $\tau' : P \to (2^A \to 2^A)$ is the so-called accumulating semantics, defined in the obvious way.

Apart from the extensional definition, little emphasis is placed on the nature of the approximate semantic function t. The dependence quads developed here can be seen as concrete representations of approximate semantic functions with more structure than merely being functions. As an analogy note that the set of linear functions between multi-dimensional real

vector spaces is included in the set of smooth functions between topological spaces. The linear functions and the associated matrix calculus nevertheless deserves special study. The added structure provides less flexibility than the totally general notion of a function, however, more information may be extracted from the concrete representations.

If $q : P \to Q_{mn}$ derives a dependence quad from a program term, its correspondence to the approximate semantic function $t : P \to (L \to L)$ is: $\mathrm{img}(x, q(p)) \approx t(p)(x)$. However, the quad $q(p)$ supports the notion of pre-image and, possibly, reverse-image, not supported by the approximate semantic function t.

Even though the section on grids defined notions of an abstraction function α_K and a corresponding concretization function γ_K, those functions are not to be confused with the abstraction and concretization functions of an abstract interpretation analysis using dependence grids as a tool. The abstraction functions of the Cousot framework map from the semantic value domain into the abstract value domain, whereas the grid abstraction serves to provide finite representations of already abstract functions. The two sets of functions serve similar purposes but at different levels.

Many program analyses can be formulated in terms of *type systems*. A (monomorphic) type system associates information about acceptable input and possible output with program functions. Polymorphic type systems can also encode dependences between inputs and outputs, as in the function type $(\alpha, \beta, \gamma) \to (\alpha, \gamma)$, where the use of the same type variable on both sides of the arrow signifies a connection between inputs and outputs. Types are usually derived from program syntax in a compositional manner by means of *type rules*. In languages without explicit type declarations, the type of a term cannot usually be computed directly by means of the specified type rules. Instead a separate *type inference* algorithm must be provided and shown correct with respect to the type rules.

The similarity between dependence algebra-based analysis and polymorphic type systems is that both frameworks provide for the encoding of dependences between inputs and outputs. But whereas this dependence is central in the approach developed here, it merely serves to parameterize types of functions so that a function with a given type may be used in different type contexts. Even a combination of polymorphism and sub-typing does not allow the simple encoding of restricted dependences between inputs and outputs provided by the dependence-based approach. On the other hand, type systems are better suited for the analysis of higher-order programs than the dependence-based approach which is best suited for analysis of first-order programs. A dependence algebra-based analysis can, however, be developed for a higher-order language as shown in Chapter 6.

Dependence algebra-based analysis also bears some resemblance to projection-based analysis [WH87, Lau87]. In projection-based analysis, the goal is to transfer representations of demands on the result of a function to representations of demands on the arguments of the function. Suppose `f` is a program function of one argument, and consider the call to `f` in the context "`f(x) + 7`". Here the demands on the results of the function could be that the result must be an integer. In projection-based analysis, a backwards map $f^{\#}$ is associated to each program function, translating demands on the result of `f` into demands on the arguments to `f`. And in particular, this backwards map is represented as a projection (a continuous, non-increasing, idempotent function on a partial order). As such, projections are closely related to the backwards part of dependence algebra-based analysis: the pre-image and the inverse image. However, dependence algebra integrates both forwards and backwards analysis, and more technically: dependence algebra allows composition, whereas the composition of two projections is not necessarily a projection.

4.13 Summary

We have introduced the notion of a dependence algebra and given the axioms for such an algebra. On top of this we defined matrices over such an algebra and extended the operations of composition, join and sum to matrices such that the set of $m \times m$ matrices over a dependence algebra forms a **DAlg**-structure as well.

Furthermore, the notion of dependence quads was built on top of the matrices, in order to model constructs that work solely as data consumers or solely as data producers. The algebra of quads was explored and equipped with sum, multiplication and join. The notion of the image of a vector through a quad will prove to be important in the following chapters.

Then the operation of concatenation of matrices and quads was defined, with the goal of modeling concatenation of data tuples in Chapter 6.

A sound finite approximation of infinite sets of quads was developed, and sound approximations of the operations on sets of quads were developed for these so-called grids.

In section 4.10 we defined a particular non-trivial construction of a **DAlg**-structure that will be used in Chapter 6 to provide abstractions of the sub-sort lattice inherent in an action semantic specification. On top of this construction we defined the notion of reverse image of a vector through a quad, a notion that will be crucial in the analysis of action semantic specifications.

Chapter 5

Trust Analysis for C

5.1 Introduction

This chapter describes the combination of the theory of dependence quadruples with the notion of trust analysis in the implementation of a trust analyzer for the C programming language [KR88]. In so doing the chapter explores the practical implementability of both of the two theoretical pillars of this thesis.

If trust analysis is to be useful in practice it must be implemented for a real life programming language. Only then can it be tested, not on toy examples, but on real programs. Also, it should be implemented for a language where there is a substantial body of existing programs for which such an analysis could be useful.

The C programming language is a real-life, mature programming language, and as such presents many practical problems from the program analysis point of view. For a program analysis to be truly useful it must deal reasonably with most of these problems. The problems include: a forgiving type system with type casts, unrestricted pointer arithmetic, pointers to functions, a context sensitive grammar due to `typedef`s, global variables shared among compilation units, functions with a variable number of arguments, arbitrary control flow (i.e., `goto`), and implementation defined evaluation order of certain expressions. In order to be useful in practice the analyzer also has to handle existing unannotated header files and libraries where the source code is not available.

The implementation deals with all of these problems in a more or less satisfactory way. Typecasts, `typedef`s, global variables, functions with a variable number of arguments, `goto`s, and implementation defined evaluation order are all handled gracefully by the current implementation.

The simplest way to introduce the concept of trust into C would be to add explicit trust types, so every variable would have to be declared either trusted or untrusted. The analyzer presented here permits such declarations, but the programmer is allowed to omit most of them. Instead the built-in functions `trust` and `check` are used at appropriate points in the program to tell the analysis that some data may from now on be trusted (presumably after appropriate validation), and that a variable must be trustworthy at a certain point. The analysis will recognize these functions and infer trust information about variables from their use. The benefit of this is that the programmer need not write as many declarations, and that the same variable can hold both trusted and untrusted values at different points in its lifetime. This may be regarded as a sort of flow-sensitivity with respect to trust. The function:

```
int f(int a, int b) {
    a = trust(b);
    check(a);
    a = distrust(b);
    ...
}
```

may be used in different places: sometimes with a trustworthy second argument, and sometimes with an untrustworthy second argument. Notice how `a` is used to hold both trusted and untrusted data.

As a good points-to analysis has not been implemented, pointers and things that they might point to have to be explicitly declared to be either trusted or untrusted. This restriction is enforced by the analyzer to make sure that it is able to determine the trustworthiness of a value obtained by a pointer dereference. The main reason for the lack of a points-to analysis is the requirement that the analysis should be modular: the analysis should be able to analyze individual source code files separately.

The present chapter first gives a short overview of the implementation, then describes the phases of the analyzer in more detail starting with the syntactic aspects of C. The internal representation of programs is described and given a formal semantics in terms of a stack machine. The notion of control dependence is defined, and an algorithm for decorating control flow graphs with control dependence information is given. Then the actual analysis working on the internal representation is described in detail, and a soundness proof with respect to the semantics of the internal representation is given. Finally a discussion and a comparison with Chapter 2 concludes.

5.2 Overview of the Implementation

The trust analyzer for C consists of several phases each of which are discussed in the following sections. This section provides an overview of the phases.

The trust analyzer is implemented as two programs: a front-end (`tcc`) written in Perl, providing the necessary glue between the trust analyzer and the real C compiler. From the users' point of view `tcc` acts as a C compiler with all the features and command line options of the underlying compiler as well as some extra options that are sent to the trust analyzer. To use the trust analyzer on a set of C programs, all the user has to do is to change his C compiler command in his `Makefile` from e.g. `gcc` to `tcc`.

The front-end is also responsible for sending the source file(s) through the C preprocessor before submitting them for analysis one at a time.

The real trust analyzer, called `tca`, is implemented in around 7000 lines of C++ code, and the phases of this program are:

- parsing, building a syntax tree, and type checking;
- compiling the syntax tree to the internal control flow graph (CFG) representation;
- computing the post-dominator tree for each function;
- decorating the control flow graph with control dependences;

- topological sorting of the static call-graph;
- trust analyzing the internal representation.

There are three reasons for this phased approach: the analyzer does not have to deal with all the problems at once; the analysis becomes faster as the hard parts can work on a tailored representation instead of source code terms, and finally, the translation to a CFG enables a uniform treatment of all kinds of unstructured local control flow.

Looked at from a sufficiently great distance the analysis can be seen as two transformations of the source program: transformation into the internal CFG form, and transformation of the CFG into a dependence quad representing the data flow of the program. The first transformation is mainly accomplished in phase two of the analyzer, whereas the second transformation takes place in the final phase. The remaining phases exist to support these transformations.

We first show a few examples of the use of the implementation. The first example illustrates the analysis of conditionals.

Example 1 Running the analyzer with the `-Tdeps` flag makes it print out the dependences between variables before and after execution of the procedure. The special `distrust` function returns its argument in untrusted form. The printout shows that `a` is found untrusted as expected, and that `b` (after `foo`) depends on `b` (before `foo`, in case `a` is false) and `c` (before `foo`), and that `b` is untrusted after `foo` (because of the indirect dependence on `a`).

```
% ./tca -Tdeps
int a,b,c;

foo() {
    a = distrust(1);

    if (a) {
        b = c;
    }
}
^D
foo:
untrusted b: b, c
c: c
untrusted a:
```

◇

Example 2 This example illustrates the inter-procedural nature of the analysis:

```
% ./tca -Tdeps
int
f(int x, int y, int z, int w)
{
    if (x) return y; else return z;
}

int r;

void
```

```
g(int a, int b)
{
    int t;

    t = f(a,b,b,t);
    r = t;
    return t;
}
^D
f:
r: r
y: y
x: x
w: w
z: z
returnvalue: y, x, z
g:
r: a, b
a: a
b: b
returnvalue: a, b
```

Here we see that the return value of `f` depends on `x`,`y`, and `z`, but not on `w`. This is used in the analysis of `g` where the return value `t` is found to depend on `a` and `b`, but not on the previous value of `t`. ◇

Example 3 In this example we show the effect of the `check` construct.

```
% ./tca -Tdeps
int y,c;

main(int x)
{
    if (c)
        check(x);
    else
        y = x;
}
^D
main:
c: c
y: c, y, x
trusted x: x
returnvalue: <out of scope>
```

The output of the analyzer should be interpreted as: `c` after `main` depends on `c` before `main`, `y` after `main` depends on `c`, `y`, and `x` before `main`, and `x` after `main` depends on `x` before `main`, and finally: `x` *before* `main` must be trustworthy (the printout format from the program is rather confusing at this point). Also, there's no `return` statement in the program, so the return value is reported as "out of scope." ◇

5.3 Parsing and Type Checking

The first phase of the analyzer parses a source code file into a syntax tree. The syntax of C requires the parser to have knowledge of types and variable scope, one cannot simply parse a program into a syntax tree and handle the types afterwards. The reason for this awkwardness is the `typedef` construct. The following program illustrates the problem:

```
int x;

foo() {
    typedef int T;
    T * x;
}

bar() {
    int T;
    T * x;
}
```

The line "`T * x;`" in `foo()` must be parsed as a declaration of the local pointer variable `x`, whereas in `bar()`, the same line must be parsed as a multiplication expression. Notice also that `typedef`s are scoped like variables: the definition of `T` in `foo()` is not visible in `bar()`.

The solution to this problem is to define types as soon as each declaration is parsed, and feed type and scope information back into the lexical analyzer. This way the parser sees different tokens for `typedef`'ed names and variables.

Another minor problem with the syntax of C is that part of the type information in a declaration is part of the variable declarator:

```
unsigned int x, *y, *z[];
```

This declares the three variables `x`, `y`, and `z` to be of type integer, pointer to integer, and array of pointers to integers, respectively. Internally in a program analyzer one would like a map from identifiers to types, so one has to parse the prefix type (`unsigned int`) and while parsing the variable declarators "extend" this prefix type to the requisite type for each variable.

There are two well-known C compilers available with source code: The GNU C compiler GCC [Sta95], and the LCC compiler by Fraser and Hanson [FH91, FH95]. It was considered to re-use the parser parts of either of these compilers in building the C analyzer, but eventually no code from these compilers was re-used. The primary reason is that in both compilers the parser is knitted closely together with the transformation to an internal representation. At no point is an entire syntax tree generated. This makes it very hard to extract just the parser parts and re-fit them for another internal representation.

Instead the parser for this project started from the free `lex` and `yacc` code available from the `comp.compilers` archives. That parser was then extended to support full ANSI C, as well as the syntax extensions particular to the trust analysis. The `yacc` code comes without semantic action code so it was necessary to write a new type-checker to go with the parser. The type checker is not as strict as it should be in a real C compiler. Type checks are only made to a degree so that subsequent phases of the analyzer won't fail. After analysis the

program is submitted to a real C compiler which will tell the user about any further type errors. The lexical analyzer is generated with the GNU `flex` tool, and the parser is generated using the `bison` parser generator.

5.3.1 Syntax Extensions

We extend the syntax of C to allow the programmer to specify global trust constraints on variables. We add the keywords `trusted` and `untrusted` as type qualifiers similar to the keywords `const` and `volatile`. With these one may write declarations such as:

```
trusted int x;
untrusted int * trusted p;
```

This declares `x` as an integer that must always hold a trusted value, and `p` as a trusted pointer to an untrusted integer.

We add three predefined functions: `trust`, `distrust`, and `check` each acting as the identity on values. `Trust` and `distrust` return their argument in trusted or untrusted form, respectively. `Check` performs an analysis-time check that the argument is trustworthy, otherwise an error message will be generated. They are used as in the following statements:

```
y = trust(x);
z = distrust(k);
w = check(a + b);
```

The first assignment informs the analysis that `y` is assigned the value of `x` and that it is trustworthy. The second assignment assigns `z` the value of `k` and informs the analysis that `z` cannot be trusted. The last assignment checks that the value of the addition of `a` and `b` is trustworthy and assigns the result to `w`. The analysis infers that for `a+b` to be trustworthy both `a` and `b` are required to hold trustworthy values at that point in the program.

All three functions are entirely analysis-time: they are removed from the program text by the preprocessor before the program is submitted to the actual compiler.

Separate analysis of program modules, and analysis of programs that use library functions with no available source code should be possible. We add some syntax allowing the programmer to annotate the prototype of a function with information about how the function acts with respect to trust. This amounts to specifying *summary information* about the internal dependences of the function.

The added prototype syntax is enough to be able to specify an arbitrary dependence quad corresponding to the function. The BNF syntax of the additions is given below. We write superscript '+' for a sequence of one or more items, and superscript '?' for optional items.

$$
\begin{array}{rcl}
\mathit{MatDeclarator} & ::= & \texttt{<<}\ \mathit{MatDecl}^{+}\ \texttt{>>} \\
\mathit{MatDecl} & ::= & \mathit{Slot}\texttt{:}\,\mathit{SlotList}^{?}\,\texttt{;} \mid \texttt{trusted}\ \mathit{Slot}\texttt{:}\,\mathit{SlotList}^{?}\,\texttt{;} \\
 & & \mid \texttt{untrusted}\ \mathit{Slot}\texttt{:}\,\mathit{SlotList}^{?}\,\texttt{;} \\
\mathit{Slot} & ::= & \mathit{BareSlot} \mid \mathit{BareSlot}\ \texttt{trusted} \\
 & & \mid \texttt{returnvalue} \mid \texttt{returnvalue trusted} \\
\mathit{BareSlot} & ::= & \mathit{Identifier} \mid \mathit{BareSlot}\texttt{.}\mathit{Identifier} \\
\mathit{SlotList} & ::= & \mathit{Slot} \mid \mathit{SlotList}\texttt{,}\,\mathit{Slot}
\end{array}
$$

A slot is either a variable or a field in a structure. Writing "`a: b,c;`" means that the value of the variable `a` after execution of the procedure depends on the values of the variables `b` and `c` before execution of the procedure. Writing "`trusted a: b,c;`" in addition means that the value of `a` is required to be trustworthy before execution of the procedure. Writing "`a: b trusted, c;`" means that there is a trusted dependence between `b` and `a` in the procedure. Writing "`untrusted a: b,c;`" means that `a` is untrusted after execution of the procedure. This notation is quite confusing, but was easy to implement.

The following example prototype informs the analysis that the function `f()` needs the global variable `x` to be trusted; and that its return value has the same trustworthiness as its argument:

```
int f(int *p) << trusted x:; returnvalue: p; >>;
```

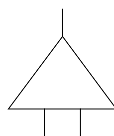
Notice that the returned C type of `f()` and the C type of the argument `p` need not be the same, the dependence encoded by "`returnvalue: p;`" does not relate directly to the C type.

5.4 The Internal Representation

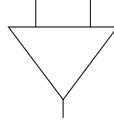
As soon as the C program is parsed and type-checked, the resulting syntax tree is compiled into the internal representation. The rich abstract syntax tree of the C program (there are more than 60 different node types) is translated into a non-standard control flow graph (CFG) with stack machine operations in the operation nodes. The control flow graph is non-standard in that all nodes have fixed in-degree and out-degree. When control paths split as in the branches of a conditional they split at a special fork node with two successors, and when control paths meet (for example after a conditional) they meet at a special join node with exactly two predecessors. This makes the control flow graph have more nodes than in a standard CFG, but only by a small constant factor (standard CFG's also have bounded out-degree, and thus the number of edges $|E| = O(|V|)$ where $|V|$ is the number of vertices in the graph). The advantage of this representation is that the algorithms become simpler as one only has to care about paths that meet in the join nodes. A depth-first traversal, for example, need only color the join nodes to avoid looping as every loop must contain a join node.

There are 6 different node types in the control flow graph:

- `StackOp`: a stack machine operation. The various stack machine operations are described separately below.
- `Fork`: a fork in the control flow graph. This is the only type of node in the CFG that has two successors. The are two variants of fork nodes: one corresponding to a conditional which pops a boolean off the evaluation stack, and an unconditional one corresponding to `goto`s. We draw a fork node as triangle with the control flow entering at the top and leaving through the two edges at the bottom.



- `Join`: a node where two control flow paths meet. This is the only kind of node that has two predecessors. A join node is drawn as a fork node up-side down with the control paths entering at the top and exiting at the bottom:



- `Call`: a node corresponding to a procedure call. It holds information about the number of parameters to pass, which procedure to call etc.
- `Fundef`: a node corresponding to a function definition. It holds information about the number of local variables, number of parameters, etc.
- `Skip`: a no-operation node which is useful during the generation of the CFG.

There is also a seventh kind of CFG node which is used for `goto`s only during the generation of the CFG, and which is replaced by a fork node in the process of fixing up `goto`s to labeled statements. Join nodes are inserted at the target of the jump to keep the in-degree of all nodes fixed.

5.5 The Stack Machine

The stack machine has the following operations.

- push: pushes the contents of a slot onto the stack.
- store: stores the top of the stack into a given slot. Leaves the stack unchanged.
- bin: a binary operation like $+$ or $-$. These are not distinguished as they must all be treated the same in the trust analysis. This operation takes the two topmost values off the stack and pushes their composition.
- pop: takes the top off the stack.
- check: checks whether the top of the stack is trustworthy, and otherwise stops the execution.
- trust: makes the top of the stack trustworthy.
- distrust: makes the top of the stack untrustworthy.

The store operation leaves the stack unchanged. This is to model the C feature that assignments are expressions and return the assigned value. Pop operations are essentially inserted at the so-called sequence points of the program, i.e., after expression statements, at the comma in a comma expression, and after the initializer and continue expression in `for` loops.

The following section more formally defines the semantics of the internal representation.

5.6 Semantics of the Flow Graph

This section presents a formal abstract semantics of the control flow graph language. The semantics is presented as a transition relation between configurations. For simplicity the semantics given here covers only local control, it does not describe calls to other procedures. The semantics for calls and returns is given later.

We must first introduce the trust-lattice Tr. It is the three point total order given as: $\mathsf{indep} \sqsubseteq \mathsf{tr} \sqsubseteq \mathsf{dis}$. The bottom point indep is used in the ensuing analysis to encode independence, and will not be used in the operational semantics below. The point tr encodes trustworthy values and dis encodes untrustworthy values.

A configuration is a triple (s, m, p) where s is a stack of trust values (trusted or untrusted), m is a memory mapping locations to trust values, and p is the program point, i.e., the current node in the CFG. Formally:

$$(s, m, p) \in \mathsf{Tr}^* \times (\mathsf{Loc} \to \mathsf{Tr}) \times \mathsf{Node}.$$

We write $t : s$ for the stack with t on the top and s being the rest of the stack. The empty stack is written $[]$.

For the purpose of presenting the semantics we represent the CFG via the following functions. The function g maps program points to node types, and the function n maps a node to its successor. For fork-nodes we define the additional function n_2 which maps the fork-node to its other successor. We write $m[l \mapsto t]$ for the memory which is like m except with location l mapped to t. The $\sqcup$ operation forms the least upper bound on the Tr lattice.

Definition 5.1 The one-step transition relation "$\mapsto$" is defined by cases as:

$$\begin{array}{rcll}
(t : s, m, p) & \mapsto & (s, m, n(p)) & \text{if } g(p) = \mathsf{pop} \\
(s, m, p) & \mapsto & (m(l) : s, m, n(p)) & \text{if } g(p) = \mathsf{push}\ l \\
(t : s, m, p) & \mapsto & (t : s, m[l \mapsto t], n(p)) & \text{if } g(p) = \mathsf{store}\ l \\
(t : t' : s, m, p) & \mapsto & ((t \sqcup t') : s, m, n(p)) & \text{if } g(p) = \mathsf{bin} \\
(t : s, m, p) & \mapsto & (\mathsf{tr} : s, m, n(p)) & \text{if } g(p) = \mathsf{trust} \\
(t : s, m, p) & \mapsto & (\mathsf{dis} : s, m, n(p)) & \text{if } g(p) = \mathsf{distrust} \\
(\mathsf{tr} : s, m, p) & \mapsto & (\mathsf{tr} : s, m, n(p)) & \text{if } g(p) = \mathsf{check} \\
(t : s, m, p) & \mapsto & (s, m, n(p)) & \text{if } g(p) = \mathsf{fork} \\
(t : s, m, p) & \mapsto & (s, m, n_2(p)) & \text{if } g(p) = \mathsf{fork} \\
(s, m, p) & \mapsto & (s, m, n(p)) & \text{if } g(p) = \mathsf{join}
\end{array}$$

□

As usual we write $\mapsto^*$ for the reflexive transitive closure of the $\mapsto$ relation. Note the non-determinism introduced by the two rules for fork-nodes. This is due to the abstract nature of the semantics. The "real" values such as integers and booleans have already been abstracted away leaving only the trust tags.

Fork operations pop a value off the top of the stack, namely the trustworthiness of the condition in the corresponding `if` statement. Unconditional `goto`s are encoded by pushing a dummy, trusted, value on the stack before forking.

A configuration is called *stuck* if it has no successor configurations. This can happen if for example the stack runs empty before time or if an untrusted value is checked.

The next sections describe how the C program is compiled into the internal CFG representation.

5.7 Compiling to Internal Representation

This phase of the analyzer translates the syntax tree of the C program into the internal CFG representation described in the previous sections.

Each function body is traversed in a depth-first manner, and a flow graph is constructed by gluing parts of graphs corresponding to sub-expressions and statements together. After the initial flow graph has been constructed, control flow edges corresponding to `goto`s are added.

In the following some examples of the translation of typical program constructs are given. First an example of the translation of a simple program to a flow graph is shown.

The trivial assignment program on the left is translated to the code sequence on the right. The numbers on the right are node numbers.

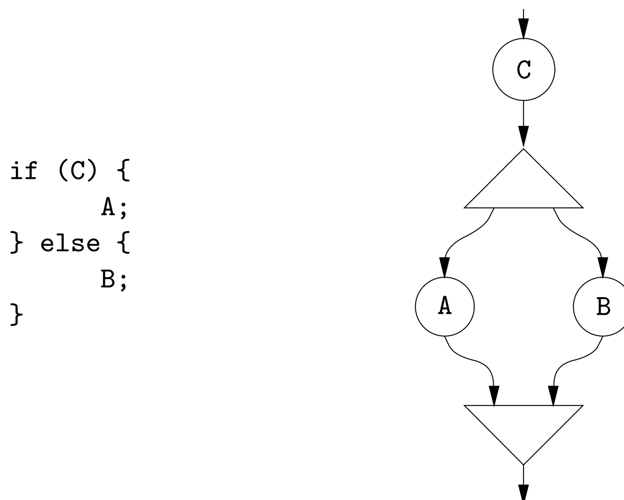
```
int a,b;
void f()
{
   a = b;
}
```

```
2: f() 0 {   ; Fundef node
1: skip      ; dummy place holder
3: push 4 0  ; push value of b
4: push 1 0  ; push trust of &a
5: trust 0 0
6: check 0 0 ; is &a trusted?
7: pop 0 0   ; pop result of the check
8: store 3 0 ; store b in a
9: pop 0 0   ; pop result of assignment
0: skip      ; dummy place holder
}
```

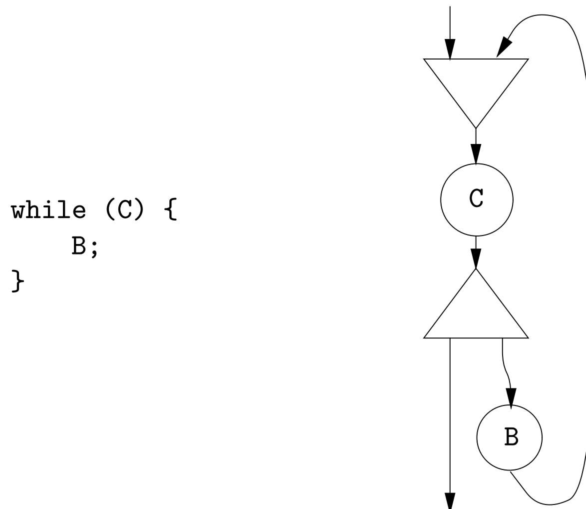
Notice how the address stored into must always be checked to be trustworthy. Just as in Chapter 2, storing into an untrustworthy address may render the entire memory untrustworthy. We later show how a peephole optimizer may eliminate many of these trivial checks.

The conditional on the left is translated to the CFG on the right:

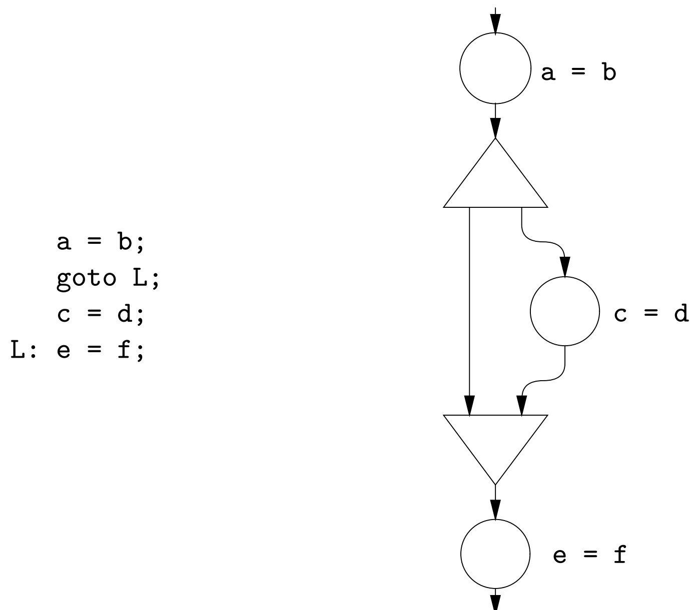
```
if (C) {
      A;
} else {
      B;
}
```



The generic while-loop program is compiled into a cyclic graph as below:



The following example shows the graph generated from a program with a `goto` statement:



A `goto` statement is compiled into a fork node with one branch pointing to the target statement and the other (never taken) branch pointing to the next statement, effectively converting every `goto` into a conditional statement. The same approach is taken in [BH93, CF94, Agr94] with application to program slicing. The technical reason for this is that we must make sure that there is always a path from any node in the body of a function to the end-node of that function. This is needed for the notion of a post-dominator to be well-defined. Post-dominators are computed later to determine the "range of influence" of conditionals.

As in the preceding example this encoding of `goto` may lead to imprecision because the graph includes control paths that do not occur in the real program. In the example, the assignment `c = d` is dead, but the analysis does not exploit this fact. Some imprecision is inevitable in a static program analysis, but more precision could be obtained here by eliminating dead code before `goto`s are converted to fork nodes. Also, a `goto` to a node that is already connected to the end node could be represented without a `fork` node.

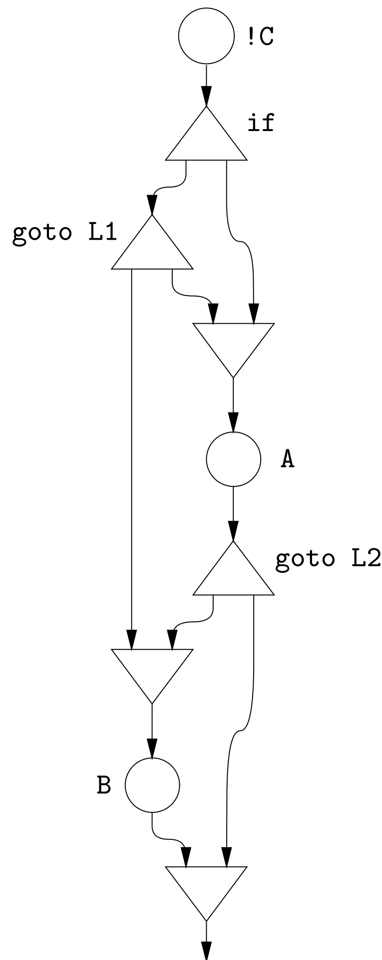
As a side effect of the handling of gotos, a structured if statement and its equivalent goto-using counterpart are not translated into equivalent graphs. This was also observed by Ball and Horwitz [BH93]. Consider the structured program

```
    if (C) { A; } else { B; }
```

and its unstructured counterpart:

```
    if (!(C)) goto L1;
    A;
    goto L2;
L1: B;
L2:
```

The graph of the structured variant was given above, whereas the graph of the unstructured variant looks like this:



Note how the CFG generated from the unstructured code contains a path from A to B whereas the structured program does not. This makes the analysis more precise on the structured code than on the unstructured code.

5.7.1 Peephole Optimization

Compiling the program to the internal control flow graph representation often yields trivial sequences of code that may be traversed many times during the analysis. As each step in the analysis is relatively expensive, it pays off to reduce the trivial instruction sequences before the analysis. Consider again the following trivial program:

```
int a,b;
main() {
        a = b;
}
```

In the previous section the long sequence of stack machine code generated from this program was shown. We show it again below for ease of reference:

```
2: main() 0  ; the function header
1: skip      ; initial body of function
3: push 3 0  ; push the value of b
4: push 1 0  ; push the constant TRUSTED: the address of a
5: trust 0 0
6: check 0 0 ; check the assigned address is trusted
7: pop 0 0   ; remove relics of address check from stack
8: store 4 0 ; store the value of b in a
9: pop 0 0   ; pop the value of b
0: skip      ; final skip
```

In this case it doesn't take much analysis to see that checking the trustworthiness of the assigned address is trivial. Our peephole optimizer does a single pass over the code sequence: eliminates most `skip` instructions, collapses the `trust` followed by the `check`, which then yields a `push` immediately followed by a `pop` which can then be collapsed, and we end up with the shorter sequence:

```
2: main() 0   ; the function header
1: skip
3: push 3 0   ; push the value of b
8: store 4 0  ; store the value of b in a
9: pop 0 0    ; pop the value of b
0: skip
```

The peephole optimizer is implemented as part of the code generation routines in the translation to internal representation, thus with optimization switched on, the lengthy code sequence is never generated in full, as the code sequence is optimized on the fly while being emitted. This saves space and an extra pass over the CFG.

5.7.2 Undetermined Evaluation Order

Binary operators and assignment have undetermined evaluation order in C. This, together with assignments and side-effecting procedure calls being expressions, means that the ordering of side effects is sometimes undetermined as in the examples below:

```
x = (y = trust(z)) + y;
arr[i = j] = i;
f(x, x = y);
x = g(y) + h(y);
```

In the first assignment the two sides of the + operator can be evaluated in any order, so the trustworthiness of `x` may or may not depend on the trustworthiness of `y` before the assignment. In the second assignment, `i` may be assigned both before and after the assignment to the array. In the call to `f()`, the assignment to `x` may be done before or after the first parameter is passed to `f()`, as evaluation order of function arguments is undetermined in C.

In the last line, the call to `g()` may define a global variable used by `h()`, and in this case the result may depend on whether `g()` is called before `h()` or vice versa.

It is usually considered bad programming practice to write C programs that rely on a certain evaluation order, as a particular ordering depends on the compiler used.

Expressions that may give different results depending on evaluation order also present a problem for a program analyzer. Consider the analysis of the argument expressions in a function call. When analyzing a particular argument, the analysis has to assume that any set of the other arguments may or may not have been evaluated before this argument. A sound way of coping with this is to analyze all possible permutations of the arguments and take the least upper bound of the dependence quads for each argument. This is of course very time consuming, as there are $n!$ permutations of n arguments.

The trust analyzer therefore implements an analysis which finds expressions that depend on evaluation order and signals an error if such an expression is found. This both helps prevent bad programming practice and enables a more precise trust analysis.

The evaluation-order-dependence analysis works interleaved with the compilation to internal flow graph form. Each expression tree is traversed in a bottom-up fashion, computing two sets of slots: a set of used slots in the (sub-) expression, and a set of defined slots in the (sub-) expression.

For each binary operator with undetermined evaluation order, the analysis checks whether the set of used slots in the left expression intersects with the set of defined slots in the right expression, and vice versa. If one of these two intersections is non-empty it means that a slot (variable) is used on one side of the binary operator and defined on the other side, which means that the value of that slot may depend on the evaluation order of the binary operator. Whenever such an evaluation-order dependence is found an error message is generated. The used slots of the binary expression is the union of the used slots of the two sub-expressions, and likewise for the set of defined slots. Pointers (and slots they may point to) have fixed trustworthiness: this takes care of situations where the slots loaded from and stored into are not known at analysis-time.

The evaluation order of the left and right hand sides of an assignment is also undetermined, so an approach similar to the binary operators is used to find evaluation order dependences. The used slots in an assignment is the union of the used slots in the two sub-expressions of the assignment. The defined slots of the assignment is the union of the defined slots of the two sub-expressions plus the slot(s) defined by the assignment. Multiple slots may be defined at once in structure assignments.

For function calls we compute the sets of used and defined slots for each argument and check that the slots used in one argument do not intersect any of the defined slots of the other arguments. This procedure requires a quadratic number of set intersections, and since

bit vectors are used to represent the sets, it is a cubic check for each function call. It sounds pretty expensive, but it seems to work in practice, as most function calls have few arguments, and usually involve few variables. Function calls with side-effecting arguments are actually very rare in real-life programs. This means that for most calls, the sets of defined slots will be empty, providing quick intersection checks.

The called function may also have side effects of its own. For each function the analysis computes an approximation of the set of globals that are used and defined by the body of the procedure, independent of the call-sites. For recursive functions fixed points for the used and defined sets are computed.

Calls to external functions are assumed to be able to define and use all global variables. Unfortunately this disallows expressions like "`f(x) + g(y)`" where both `f()` and `g()` are external, as they might side-effect global variables in their own file-scope.

Example 4 Illustration of the evaluation order dependence analysis in action:

```
% ./tca
int x,y,z,i,j;
trusted int arr[34];
void f(int, int);
main() {
    x = (y = trust(z)) + y;
    arr[i = j] = i;
    f(x, x = y);
}
^D
tca:<stdin>:5: evaluation order dependence in binary operator.
tca:<stdin>:6: evaluation order dependence in assignment.
tca:<stdin>:7: evaluation order dependence in call to 'f',
               parameters 1 and 2.
tca: Errors detected, bailing out!
```

◇

It is also possible to evade the problem entirely by assuming a fixed evaluation order, as is done in the analyses in C-mix [And94].

5.8 Control Dependence and Decoration

An assignment made under untrusted control should make the assigned slot untrustworthy. Consider the example

```
x = distrust(y);
if (x) {
     z = w;
}
```

Because the condition in the `if` statement is untrustworthy we cannot trust whether the assignment to `z` is made. Thus even though `w` may be trustworthy we shouldn't be able to trust the value of `z` after the assignment.

Extending this notion to unstructured control flow is more complicated. In the above example it is clear that no assignments made after the `if`-block are influenced by the untrusted condition, they will be executed no matter what the value of `x` is. Now consider adding a `goto` in the body of the `if`:

```
   x = distrust(y);
   if (x) {
        z = w;
        goto L;
   }
   a = b;
L: c = d;
```

It is clear that the assignment to `a` is made only when `x` is false, and it is thus under the control of `x`, whereas the assignment to `c` is always executed.

Example 5 Listed below is the internal representation of the above program containing the `goto`. Notice how the nodes between nodes 11 and 24 are marked control dependent on node 9.

```
 3: push 10 0          ; x = distrust(y)
 4: distrust 0 0
 5: store 6 0
 7: pop 0 0
 8: push 6 0           ; if (x)
 9: fork  (11, 10) IPD = 24
11: 9 push 7 0         ; z = w
12: 9 store 4 0
14: 9 pop 0 0
15: 9 fgoto  (10, 24) IPD = 24
10: 9 join
16: 9 push 3 0         ; a = b
17: 9 store 5 0
19: 9 pop 0 0
24: 9 join
20: push 8 0           ; c = d
21: store 9 0
23: pop 0 0
```

Notice also how nodes 16, 17, and 19 are *not* marked control dependent on node 15, the `goto` node. This is an optimization as the "condition" in `goto` `fork`-nodes is always trusted (the branch is unconditionally taken). ◇

In the following we introduce general tools that will enable detection of the extent of control influences in the presence of unstructured control flow. We first need to introduce some notation.

Definition 5.2 A *path* in a graph is a sequence of vertices. Paths are also considered as sets of vertices, so if the vertex X belongs to the path P we write $X \in P$. We write $P : [X \to Y]$ to say that P is a directed path from X to Y including both X and Y. □

The concept of *post-dominators* [Pro59, LM69, All70, ASD86] has proved to be important for determining (a safe approximation of) the extent of the control influence of a conditional.

Definition 5.3 Let (V, E) be a directed graph with a designated end-node N such that N is reachable from all nodes in V. A node $Y \neq X$ *post-dominates* a node X if all paths from X to N pass through Y. In symbols we write:

$$\mathrm{pd}(X, Y) \iff X \neq Y \text{ and } \forall P : [X \to N] : Y \in P.$$

□

If a node post-dominates a fork node in the CFG then that node is clearly not control dependent on the fork node, as it will be executed regardless of the branch taken by the fork node.

As a matter of efficiency we don't really need to compute the entire post-dominator relation which is quadratic in the size of the graph, instead we compute immediate post-dominators. An immediate post-dominator of a node is the post-dominator "closest" to that node.

Definition 5.4 Let S be the set of post-dominators of X. Each node in S is by definition on *all* paths from X to the end-node so there is a unique node in S which is met first going from X to N. This node is called the *immediate post-dominator* of X, and is written $\mathrm{IPD}(X)$. □

Imagine drawing an edge between each node and its immediate post-dominator. This creates an inverse tree rooted at the end-node. This tree is called the post-dominator tree.

The post-dominator tree of a control flow graph can be computed in linear time [Har85] by a complicated tuning of the algorithm of Lengauer and Tarjan [LT79]. The algorithm used in the implementation is the simpler $O(n \log n)$ algorithm from [LT79].

The use of post-dominators for determining the extent of control dependences has been widely used. Ferrante et al. [FOW87] use it for computing the so-called program dependence graph used in program slicing. Denning suggests it to handle unstructured control flow in a security analysis [Den82], and Andersen [And94] uses it in the C-mix partial evaluator to handle control dependences.

5.8.1 Decoration

Having computed the post-dominator tree for a function body we then decorate all the flow graph nodes with a list of the fork nodes on which they are control dependent.

The definition of control dependence given by Ferrante et al. [FOW87] (also used in [BH93, CF94, And94]) is the following.

Definition 5.5 [Ferrante et al.] Let G be a CFG. Let X and Y be nodes in G. Y is *control dependent* on X iff:

1. there exists a directed path from X to Y in G with any node Z (except X and Y) on the path post-dominated by Y, and

2. X is not post-dominated by Y.

In symbols we write $\mathrm{CDF}(X, Y)$ whenever Y is control dependent on X according to this definition. □

The above notion of control dependence is intransitive. The body of a conditional statement is control dependent only on the immediately enclosing conditional, not on any other enclosing conditionals. In the program:

```
if (A) {
      if (B) {
           C;
      }
}
```

the statement `C` is control dependent on the `B` conditional, it is not directly control dependent on the `A` conditional. The inner conditional is, however, control dependent on the `A` conditional.

The Ferrante-Ottenstein-Warren algorithm [FOW87] for computing control dependences from a CFG and its post-dominator tree can roughly be described as consisting of the two steps:

- Compute the set S of directed edges (A, B) in the CFG such that B is not a post-dominator of A.
- For each edge (A, B) in S trace backwards in the post-dominator tree from B to A's parent, marking all nodes visited as control dependent on A.

The set S can be computed essentially for free during the construction of the post-dominator tree. The total cost of their algorithm is $O(n^2)$ where n is the number of edges in the CFG. Cytron, Ferrante and Rosen [CFR$^+$89, CFR91] present a better algorithm based on dominance frontiers, but with the same worst-case complexity.

However, as stated above, their algorithm computes intransitive control dependences, and we need the transitive closure of these dependences. Instead of using the Ferrante-Ottenstein-Warren algorithm augmented with a transitive closure, we use a more iterative algorithm that improves sharing of the lists of control dependences.

We write $\mathrm{CDF}^+(X, Y)$ for the transitive closure of the above relation, formally:

$$\mathrm{CDF}^+(X, Y) \iff \mathrm{CDF}(X, Y) \text{ or } \exists W : \mathrm{CDF}^+(X, W) \text{ and } \mathrm{CDF}(W, Y).$$

Several characterizations of transitive control dependence have been given in the literature. Beck, Johnson and Pingali [BJP91] prove that two nodes X and Y are in the transitive control dependence relation (i.e., $\mathrm{CDF}^+(X, Y)$) if and only if Y is *between* X and its immediate post-dominator, where between-ness is defined as:

Definition 5.6 The node Y is *between* the node X and its immediate post-dominator $\mathrm{IPD}(X)$ whenever there is a non-empty path from X to Y that does not include $\mathrm{IPD}(X)$. □

In [Wei92] Weiss proves that $\mathrm{CDF}^+(X, Y)$ if and only if X is *relevant* to Y, where a node X is *relevant* to the node Y whenever there are non-empty paths $P : [X \to Y]$ and $Q : [X \to N]$ such that $P \cap Q = \{X\}$. We will, however, not use this characterization here.

5.8.2 Computing Control Dependences

We compute control dependences with respect to Definition 5.6. We traverse the CFG along the control flow edges carrying along a set of fork nodes that we have met so far and whose post-dominator we haven't met. Whenever a post-dominator of a fork node in the set is encountered, we remove the fork node from the set. To ensure termination each node is decorated with the union of all the sets that were current when the node was reached.

A high-level, pseudo-code algorithm for decorating the CFG is given below. Let $G = (V, E)$ be the CFG and let $S \in V$ be the start node. We assume all nodes Y are initially decorated with the empty set, $\mathsf{decor}[Y] = \emptyset$.

```
decorate(Y, D) =
    D' := D − {X ∈ D | Y = IPD(X)}
    if D' ⊈ decor[Y] then
        decor[Y] := D' ∪ decor[Y]
        if Y is a fork node then
            D'' := decor[Y] ∪ {Y}
        else
            D'' := D'
        end if
        for all successors Z of Y decorate(Z, D'')
    end if
end.
```

The decoration is initiated by the call $\mathsf{decorate}(S, \{\star\})$ where $\star$ is a special node not in the graph, just to get the algorithm started. The special node is subsequently disregarded.

Proposition 5.7 *The* decorate *algorithm is correct, that is, after it terminates:*

$$\mathsf{decor}[Y] = \{X \in V \mid \mathrm{CDF}^+(X, Y)\}.$$

It is not hard to see that the array decor grows monotonically with respect to set-inclusion, and that each $\mathsf{decor}[Y]$ cannot grow to more than $|V|$. So the algorithm terminates.

The following invariant holds at the start of each invocation of the procedure:

1. For all $Y \in V$ we have:

$$\mathsf{decor}[Y] \subseteq \{X \in V \mid X \text{ is a fork node, there is a path } P \text{ from } X \text{ to } Y, \text{ and } \mathrm{IPD}(X) \notin P\},$$

   and

2. for the parameters Y and D the following holds:

$$D = \{X \in V \mid X \text{ is a fork node, there is a path } P \text{ from } X \text{ to } Y, \text{ and } \mathrm{IPD}(X) \notin P - \{Y\}\}.$$

When the algorithm terminates we must have $D' \subseteq \mathsf{decor}[Y]$ in all the recursive calls. This together with the invariant ensures that whenever Y is between X and $\mathrm{IPD}(X)$, $X \in \mathsf{decor}[Y]$. Beck, Johnson and Pingali's result then ensures the correctness of the algorithm.

As the set $\mathsf{decor}[Y]$ must grow each time a node is visited, each node can be visited at most n times where n is the number of nodes in the graph. The cost of visiting a node once is in $O(n)$, so the worst-case complexity of the computation is in $O(n^3)$. In practice the D sets have much fewer than n elements. If the sets are implemented as simple linked lists, and the CFG is generated from structured code (the common case), the operations on the sets can be done in constant time, as fork-nodes and their immediate post-dominators are met in a stack-like manner. In this case much of the decor lists can be shared among nodes as well.

In hindsight it may not have been a good idea to compute transitive control dependences at all, but better to decorate each node with its immediate control dependences, and just follow those links on demand in the ensuing analysis. However, this yields a more complex lookup of control dependences in the expensive analysis.

5.9 The Static Call Graph

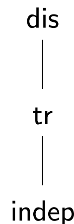
The next phase of the analysis does a topological sort of the static call graph of the program module. This is done merely to speed up convergence of the iterative analysis in the next phase.

If procedure P calls procedure Q then it is advantageous to analyze Q before P, as the result of analyzing Q is needed during the analysis of P. When mutually recursive procedures are encountered (strongly connected components in the static call graph) they are ordered arbitrarily, and the iterative analysis in the next phase analyzes each procedure “on demand” iterating until a fixed point is reached. Calls via function pointers are not treated in this phase.

5.10 The Trust Analysis Proper

This section gives a formal description of the trust analysis that happens at the CFG level. This was the part of the analyzer that proved the hardest to get right, which is why we give a soundness theorem for the analysis in this section. The translation of a C program into the CFG and stack machine is pretty standard, and therefore does not warrant a huge formal treatment.

For each procedure the analysis computes a quad over the **DAlg**-structure $(\mathsf{Tr}, \sqcup, \sqcup, \sqcap, \mathbf{0}, \mathbf{1})$ with $\mathbf{0} = \mathsf{indep}$, and $\mathbf{1} = \mathsf{dis}$, built from the lattice Tr with the ordering $\sqsubseteq$:



The purpose of this phase of the analysis is twofold: to determine the data dependences of the program, and to reject programs that the analysis cannot guarantee not to go wrong. Thus there is a strong similarity to type inference. The purpose of type inference is to determine a type for a program, and reject untypable programs. To quote the slogan from [WF94]: “well-typed programs do not go wrong.”

With reference to the semantics of the stack machine given in Sect. 5.6 we characterize programs that go wrong as programs that end up in the following stuck configuration:

$$(\mathsf{dis} : s, m, p) \text{ where } g(p) = \mathsf{check}.$$

It is not the above configuration that is wrong as such, but the fact that there is no possible successor configuration. That is, the error does not happen until a transition from the above configuration is attempted. We therefore add the new configuration WRONG to the set of possible configurations, and extend the transition relation with the rule:

$$(\mathsf{dis} : s, m, p) \mapsto \mathsf{WRONG} \text{ if } g(p) = \mathsf{check}.$$

The exact class of programs that end up in such a configuration is of course undecidable, and as in other program analyses we aim for a sound computable approximation of the complement of this class. Also note that there are several other stuck configurations of the stack machine, especially of the kind where the stack contains too few items for an operation. We do not concern ourselves with these stuck configurations here, as they will not be present in the code generated by the compilation of the original C program.

Having defined what is meant by a program going wrong we must then define what is meant by "well-typed", and make sure that the well-typed programs cannot go wrong.

To each reduction sequence in the original semantics the analysis computes a quadruple (s, A, t, u) encoding the dependences between the memory of the start configuration and the memory of the end configuration. A reduction sequence is called "well-typed" whenever the u component is $\mathbf{0}$, that is, when there is no dependence between the inherently untrustworthy src slot and the sink slot, which is required to be always trustworthy.

The t vector of the quad encodes demands on the input memory. Think of the sink slot as a special slot that must always be trusted. If there is a $\mathbf{1}$-dependence between a memory slot containing an untrusted value and sink it means that the memory doesn't satisfy the demands on the input, and all bets are off, that is, starting a well-typed program from a configuration that doesn't satisfy the demands of t may well go wrong.

One can look at each column of the matrix of a quad as encoding the set of locations that the location corresponding to the column is dependent upon, except that both trusted and untrusted dependences are handled by the quad representation.

5.10.1 The Analysis

Before we can state the formal definition of the analysis and its soundness we need some definitions and notation.

Definition 5.8 For a vector v we write v^t for its *transpose.* □

Definition 5.9 Define the *injection row-vector* J_k as

$$J_k = (\mathbf{0}, \ldots, \mathbf{0}, \mathbf{1}, \mathbf{0}, \ldots, \mathbf{0}),$$

where the sole $\mathbf{1}$ entry is in the k'th position. By abuse of notation we also write J_k for the $1 \times m$ quad $(\mathbf{0}, J_k, \mathbf{0}, \mathbf{0})$. □

Fact 5.10 *Let m be a row-vector then extracting the k'th element from the vector can be done by means of J_k:*

$$m_k = mJ_k^t.$$

Definition 5.11 Define the square matrix I_k as the identity matrix except that the (k,k)'th entry is $\mathbf{0}$. □

To keep track of the dependences on the stack during analysis we define a reduction system between abstract configurations.

For the trust and distrust operations we need to modify the dependences on top of the stack. This uses the two quads T and D defined below.

Definition 5.12 Define the 1×1 quad T as:

$$T = (\mathsf{tr}, \mathsf{tr}, \mathbf{0}, \mathbf{0}),$$

and define the 1×1 quad D as:

$$D = (\mathsf{dis}, \mathsf{dis}, \mathbf{0}, \mathbf{0}).$$

□

If we view a memory m as a row-vector of trust-values we can formulate an update by matrix operations.

Lemma 5.13 *Let m be a memory (row-)vector, l a location (i.e., an index in the memory vector, we write $m(l)$ for m_l), and let x be a trust value. Then*

$$m[l \mapsto x] = mI_l + xJ_l.$$

Proof. By definition of I_l and J_l. □

Lemma 5.14 *Let m be a row-vector, $l_1, \ldots, l_n$ be distinct locations, and let $x_1, \ldots, x_n$ be trust values. Then*

$$m[l_1 \mapsto x_1, \ldots, l_n \mapsto x_n] = m \prod_i I_{l_i} + \sum_i x_i J_{l_i} \ .$$

Proof. As the locations are distinct we have

$$m[l_1 \mapsto x_1, \ldots, l_n \mapsto x_n] = m[l_1 \mapsto x_1][l_2 \mapsto x_2] \ldots [l_n \mapsto x_n].$$

By the previous lemma this is the same as:

$$(\cdots((mI_{l_1} + x_1 J_{l_1})I_{l_2} + x_2 J_{l_2})\cdots)I_{l_n} + x_n J_{l_n}.$$

By distributivity, and because all the l_i are distinct this is:

$$mI_{l_1} I_{l_2} \cdots I_{l_n} + x_1 J_{l_1} + x_2 J_{l_2} + \cdots + x_n J_{l_n}.$$

□

Definition 5.15 Let M be the number of variables. Let S be a stack of $M \times 1$ quads, Q a $M \times M$ quad, and p a program point in the CFG. An *abstract configuration* is a triple (S, Q, p). We write $V : S$ for the stack with V on top and S being the rest of the stack. □

Re-using notation from the concrete semantics of the stack machine, we can now define the abstract reductions.

Definition 5.16 The *abstract reduction* relation is defined by the following rules. For brevity we write Q for the $M \times M$ quad (s, A, t, u), and V for the $M \times 1$ quad (s_V, A_V, t_V, u_V).

$$\begin{array}{rcll}
(S, Q, p) & \rhd & ((sJ_l^t, AJ_l^t, \mathbf{0}, \mathbf{0}) : S, Q, n(p)) & \text{if } g(p) = \textsf{push } l \\
(V : S, Q, p) & \rhd & (S, Q, n(p)) & \text{if } g(p) = \textsf{pop} \\
(V : S, Q, p) & \rhd & (VT : S, Q, n(p)) & \text{if } g(p) = \textsf{trust} \\
(V : S, Q, p) & \rhd & (VD : S, Q, n(p)) & \text{if } g(p) = \textsf{distrust} \\
(V : S, Q, p) & \rhd & (V : S, (s, A, A_V + t, s_V + u), n(p)) & \text{if } g(p) = \textsf{check} \\
(V : V' : S, Q, p) & \rhd & ((V \sqcup V') : S, Q, n(p)) & \text{if } g(p) = \textsf{bin}
\end{array}$$

and for $g(p) = \textsf{store } l$ we have:

$$(V : S, Q, p) \rhd (V : S, (sI_l + s_V J_l, AI_l + A_V J_l, t, u), n(p)).$$

□

Note that each transition can be implemented in $O(M)$ time.

5.10.2 Basic Soundness

For the soundness theorem we need to be able to state that an abstract stack faithfully models a concrete stack with a concrete memory.

Definition 5.17 An abstract stack S *models* a concrete stack c and a memory m, written $S \models m, c$ iff:

1. $|S| = |c|$, i.e., the lengths of the stacks are equal, and
2. for each abstract stack element $S_i = (s_i, A_i, t_i, u_i)$, and concrete stack element c_i we have
$$c_i \sqsubseteq \text{img}(m, S_i).$$

□

We can now formulate the basic soundness theorem for single reductions. Note that the abstract reduction relation has not yet been defined for fork and join nodes. In words, the soundness property says that: if the concrete machine takes a step and the abstract machine takes the corresponding step (it is a simple observation that this is possible), and the start memory m' of the concrete machine is safely approximated by the quad Q relative to the initial memory m, the concrete stack is safely modeled by the abstract stack, and that the resulting quad Q' is internally consistent ($u' \sqsubseteq \textsf{tr}$) and the initial memory satisfies the demands of the resulting quad, then the resulting concrete memory m'' is safely modeled by Q', and the resulting abstract stack safely models the resulting concrete stack.

In other words, Q corresponds to the change between m and m', whereas Q' corresponds to the change between m and m''. Likewise S models the stack c relative to m, whereas S' models the stack c' also relative to m.

Theorem 5.18 (Basic Soundness) *With the notation $Q = (s, A, t, u)$, $S_i = (s_i, A_i, t_i, u_i)$ for $i = 1 \dots k$, and $Q' = (s', A', t', u')$, the following implication holds. If*

$$\begin{aligned}
(c, m', p) &\mapsto (c', m'', p') && (5.1)\\
(S, Q, p) &\triangleright (S', Q', p') && (5.2)\\
m' &\sqsubseteq \mathrm{img}(m, Q) && (5.3)\\
S &\models m, c && (5.4)\\
mt' &\sqsubseteq \mathsf{tr} && (5.5)\\
u' &\sqsubseteq \mathsf{tr} && (5.6)
\end{aligned}$$

then

$$\begin{aligned}
m'' &\sqsubseteq \mathrm{img}(m, Q') && (5.7)\\
S' &\models m, c'. && (5.8)
\end{aligned}$$

Proof. By cases on the value of $g(p)$. In all cases we use the notation $V = (s_V, A_V, t_V, u_V)$.

- $\mathsf{store}\ l$. We have

$$\begin{aligned}
(V : S, Q, p) &\triangleright (V : S, (sI_l + s_V J_l, AI_l + A_V J_l, t, u), n(p))\\
(x : s, m, p) &\mapsto (x : s, m[l \mapsto x], n(p))
\end{aligned}$$

  Using Lemma 5.13, (5.4), the definition of img(), (5.3), and the definition of $\triangleright$, in that order we compute:

$$\begin{aligned}
m'[l \mapsto x] &= m'I_l + xJ_l\\
&\sqsubseteq m'I_l + (mA_V + s_V)J_l\\
&\sqsubseteq (mA + s)I_l + (mA_V + s_V)J_l\\
&= mAI_l + sI_l + mA_V J_l + s_V J_l\\
&= m(AI_l + A_V J_l) + sI_l + s_V J_l\\
&= mA' + s'.
\end{aligned}$$

  which proves (5.7). For (5.8) observe that both the concrete stack and the abstract stack are unchanged, so it follows directly from (5.4).

- $\mathsf{push}\ l$. Observe that here $m'' = m'$, and $Q' = Q$, so (5.7) is immediate from (5.3). For (5.8) we just have to show that

$$m'(l) \sqsubseteq mAJ_l^t + sJ_l^t,$$

  as the requirement on the size of the stacks is trivially fulfilled. By (5.3) we have

$$\begin{aligned}
m'(l) &= m'_l\\
&\sqsubseteq (mA + s)_l\\
&= mAJ_l^t + sJ_l^t.
\end{aligned}$$

- check. For (5.7) we have $m'' = m'$, and $m' \sqsubseteq mA + s = mA' + s' = \mathrm{img}(m, Q')$. Property (5.8) is trivial as the stacks are unchanged.

- trust. We have $m'' = m'$, and $Q = Q'$, so (5.7) clearly holds. For (5.8) we just need to show that

$$\mathsf{tr} \sqsubseteq \mathrm{img}(m, VT),$$

but by definition of T we have $\mathrm{img}(m, VT) = \mathsf{tr}$.

- distrust. Here (5.7) follows as for trust, and $\mathrm{img}(m, VD) = \mathsf{dis}$ together with (5.4) ensures (5.8).

- bin. Again we have $m'' = m'$, and $Q' = Q$ so (5.7) holds. For (5.8) we must show

$$x \sqcup x' \sqsubseteq \mathrm{img}(m, V \sqcup V').$$

Here we exploit that for this particular **DAlg**-structure l.u.b. is the same as +, so using (5.4) we may compute

$$\begin{aligned} \mathrm{img}(m, V + V') &= m(A_V + A_{V'}) + s_V + s_{V'} \\ &= \mathrm{img}(m, V) + \mathrm{img}(m, V') \\ &\sqsupseteq x + x'. \end{aligned}$$

- pop. Here we also have $Q = Q'$ and $m' = m''$, so (5.7) and (5.8) follow directly from (5.3) and (5.4).

□

We shall demonstrate that computations that go wrong are not found well-typed by the analysis. Assume $g(p) = \mathsf{check}$, and consider the transition:

$$(\mathsf{dis} : c, m', p) \mapsto \mathsf{WRONG},$$

and assume as in the soundness theorem that

$$(V : S, Q, p) \triangleright (V : S, (s, A, A_V + t, s_V + u), n(p)),$$

$mt' = mA_V + mt \sqsubseteq \mathsf{tr}$, and $u' = s_V + u \sqsubseteq \mathsf{tr}$. In particular this means that $s_V \sqsubseteq \mathsf{tr}$. From $V : S \models m, (\mathsf{dis} : c)$ we get $\mathsf{dis} \sqsubseteq \mathrm{img}(m, V) = mA_V + s_V$. So we must have $mA_V = \mathsf{dis}$, and thus $mt' = \mathsf{dis}$, and m therefore doesn't satisfy the demands of Q' and the soundness theorem doesn't guarantee anything.

Note that contrary to the notion of soundness discussed in Chapter 4, the previous theorem does not involve the pre-image function at all. The motivation for using the pre-image inequality ($m \sqsubseteq \mathrm{pre}(Q, m)$) would be to infer demands on the start memory from the quad and the end memory, but the algebra used here does not facilitate meaningful demands to be propagated in this fashion.

5.10.3 Sequences

Soundness of the analysis for sequences of reductions is a simple induction argument. We write $\mapsto^*$ for the reflexive and transitive closure of the $\mapsto$ relation, and likewise for the $\triangleright$ relation. Before the actual soundness theorem we need a little lemma to show that demands on input never decrease during execution.

Lemma 5.19 *If* $(S,(s,A,t,u),p) \triangleright^* (S',(s',A',t',u'),p')$ *then* $t \sqsubseteq t'$ *and* $u \sqsubseteq u'$*.*

Proof. By inspection of the rules defining the $\triangleright$ relation, and by induction on the length of the reduction sequence, using that the $\sqsubseteq$ ordering is transitive. □

Theorem 5.20 (Transitive Soundness) *If*

$$
\begin{aligned}
(c,m',p) &\mapsto^* (c'',m''',p'') && (5.9)\\
(S,Q,p) &\triangleright^* (S'',Q'',p'') && (5.10)\\
m' &\sqsubseteq \mathrm{img}(m,Q) && (5.11)\\
S &\models m,c && (5.12)\\
mt'' &\sqsubseteq \mathsf{tr} && (5.13)\\
u'' &\sqsubseteq \mathsf{tr} && (5.14)
\end{aligned}
$$

then

$$
\begin{aligned}
m''' &\sqsubseteq \mathrm{img}(m,Q'') && (5.15)\\
S'' &\models m,c''. && (5.16)
\end{aligned}
$$

where $Q=(s,A,t,u)$, $S_i=(s_i,A_i,t_i,u_i)$ *for* $i=1\ldots k$, *and* $Q''=(s'',A'',t'',u'')$.

Proof. By induction on the reduction sequence. For the base case we have $p=p''$, $c=c''$, $m'=m'''$, and $Q=Q''$ so the theorem holds trivially.

Consider the reduction sequences (concrete and abstract):

$$
\begin{array}{lclcl}
(c,m',p) & \mapsto & (c',m'',p') & \mapsto^* & (c'',m''',p''),\\
(S,Q,p) & \triangleright & (S',Q',p') & \triangleright^* & (S'',Q'',p'').
\end{array}
$$

By Lemma 5.19 we have $t' \sqsubseteq t''$ and $u' \sqsubseteq u''$, and thus $mt' \sqsubseteq mt'' \sqsubseteq \mathsf{tr}$, and $u' \sqsubseteq u'' \sqsubseteq \mathsf{tr}$. So the prerequisites for Theorem 5.18 are fulfilled, and we get

$$
\begin{aligned}
m'' &\sqsubseteq \mathrm{img}(m,Q'),\\
S' &\models m,c'.
\end{aligned}
$$

We can now apply the induction hypothesis to obtain the desired result. □

The following theorem allows separate analysis of code sequences, such as procedure bodies. Intuitively the theorem says that a procedure body can be analyzed by the abstract machine *once* starting from the identity quad and an empty stack. The quad Q_1 resulting from this analysis is saved. Whenever a call to the procedure is encountered the saved quad is multiplied onto the current quad in the abstract machine and the result is a sound approximation of the memory after the call. The actual theorem is a little more general in order to facilitate the proof. A little notation: for an abstract stack $S=V_1:V_2:\ldots:V_n$, and a quad Q we write $QS=QV_1:QV_2:\ldots:QV_n$.

Theorem 5.21 *Let* $Q = (s, A, t, u)$, *and* $Q_1 = (s_1, A_1, t_1, u_1)$. *If*

$$
\begin{aligned}
(c, m', p) &\mapsto^* (c', m'', p') &(5.17)\\
(S_0, \mathrm{Id}, p) &\triangleright^* (S_1, Q_1, p') &(5.18)\\
S_0 &\models m', c &(5.19)\\
m' &\sqsubseteq \mathrm{img}(m, Q) &(5.20)\\
m' t_1 &\sqsubseteq \mathsf{tr} &(5.21)\\
u_1 &\sqsubseteq \mathsf{tr} &(5.22)
\end{aligned}
$$

then

$$
\begin{aligned}
m'' &\sqsubseteq \mathrm{img}(m, QQ_1)\\
QS_1 &\models m, c'
\end{aligned}
$$

Proof. We always have $m' \sqsubseteq \mathrm{img}(m', \mathrm{Id})$, so by the previous soundness theorem we have:

$$
\begin{aligned}
m'' &\sqsubseteq \mathrm{img}(m', Q_1)\\
S_1 &\models m', c' .
\end{aligned}
$$

By the definition of $\models$ this means that $|S_1| = |c'|$, for all $i \leq |S_1|$ we have $c'_i \sqsubseteq \mathrm{img}(m', V_{1i})$, where $S_1 = V_{11} : V_{12} : \ldots : V_{1|S_1|}$. By (5.20) and Theorem 4.16 we then have:

$$m'' \sqsubseteq \mathrm{img}(\mathrm{img}(m, Q), Q_1) = \mathrm{img}(m, QQ_1),$$

and likewise for the stack:

$$c'_i \sqsubseteq \mathrm{img}(\mathrm{img}(m, Q), V_{1i}) = \mathrm{img}(m, QV_{1i}),$$

which shows the required result. □

So far the soundness theorems have concerned only straight line code. We define the behavior of the abstract machine on fork and join nodes in analogy with the concrete stack machine:

$$
\begin{aligned}
(S, Q, p) &\triangleright (S, Q, n(p)) &&\text{if } g(p) = \mathsf{fork}\\
(S, Q, p) &\triangleright (S, Q, n_2(p)) &&\text{if } g(p) = \mathsf{fork}\\
(S, Q, p) &\triangleright (S, Q, n(p)) &&\text{if } g(p) = \mathsf{join}
\end{aligned}
$$

It is not hard to see that if there are two possible paths between program points p and p' in the CFG for the concrete machine, then the same paths can be taken by the abstract machine. Also, if we form the join of the quads computed by the abstract machine for each path we get a sound approximation of both of the concrete paths, that is, the analysis can compute the "join over all paths" to get a sound approximation of the data flow for all possible executions of the concrete machine. As the lattice of quads has finite height we can compute least fixed points to approximate the data flow along looping paths.

Because of the non-deterministic nature of the concrete stack machine we cannot express control dependences formally. As there are no actual values in the concrete machine, the fork nodes act independently of the trust of the condition. The implementation of the analysis handles control dependences by for each fork node remembering the top of the S stack (the

condition), and for each store operation traversing the list of fork nodes that the store is control dependent upon and adding the dependences of the saved conditions to the dependences of the stored value.

More formally: Let F be a map from fork nodes to $1 \times M$ vectors, remembering the trustworthiness of the condition. Whenever a configuration of the form $(V : S, Q, X)$ is encountered by the abstract machine, where X is a fork node, the map F is updated to $F[X \mapsto F(X) \sqcup V]$. Whenever a configuration of the form $(V' : S', Q', Y)$ is encountered, where Y is a store node, the V' part of the stack is updated to $V' \sqcup \bigsqcup_{\{X \mid \mathrm{CDF}^+(X,Y)\}} F(X)$ before execution of the store operation proceeds.

5.10.4 Calls and Returns

So far this section has only treated intra-procedural trust analysis; calls to other functions have been ignored. We first need to extend the semantics of the concrete stack machine. To this end we define an augmented transition system where configurations consist of the ordinary concrete configurations together with a control stack C and a current CFG g:

$$((s, m, p), C, g).$$

We write $\mapsto_g$ to make the CFG g explicit in the $\mapsto$ relation. A program function f is regarded as a pair $(g', (a_1, \ldots, a_n))$ where g' is the CFG of the function body, and $a_1, \ldots, a_n$ identifies the distinct slots used for the n arguments. We do not treat functions with a variable number of argument here, even though the implementation handles such functions. The notation "entry f" denotes the entry node of the CFG for f.

The control stack C is made up of triples (g, p, c) where g is the CFG of the calling function, p is the successor of the call node in that CFG, and c is the saved evaluation stack of the caller. The transition relation $(\Rightarrow)$ for this system is defined by the following rules.

$$\frac{(c, m, p) \mapsto_g (c', m', p')}{((c, m, p), C, g) \Rightarrow ((c', m', p'), C, g)}$$

$$\frac{g(p) = \mathsf{call}\ f, \quad f = (g', (a_1, \ldots, a_n)), \quad c = c_1 : \ldots : c_n : c'}{((c, m, p), C, g) \Rightarrow (([], m[a_1 \mapsto c_1, \ldots, a_n \mapsto c_n], \mathsf{entry} f), (g, n(p), c') : C, g')}$$

$$\frac{g(p) = \mathsf{return}}{((x : c, m, p), (g', p', c') : C, g) \Rightarrow ((c', m[l_\mathrm{R} \mapsto x], p'), C, g')}$$

Note that each call does not create new local variables for the called procedure. In effect all locals are treated as `static` variables, that is, all calls to the same procedure are treated equally regardless of parameter values: the analysis is monovariant.

Also notice that results are returned from functions by storing into a designated location (l_R) from which it can be pushed onto the stack. This means that the function call "`x = f(a,b)`" is compiled into the sequence:

```
push b
push a
call f
push res
```

```
store x
pop
```

Note that this does not rule out functions returning both trusted and untrusted values, as the slot (l_{R}) is not associated with an unchanging trustworthiness.

The abstract stack machine is augmented with a control stack component and a CFG component in the same fashion as the concrete machine: An augmented abstract configuration is written as $((S,Q,p),A,g)$ where (S,Q,p) is an (intra-procedural) configuration for the stack machine, A is the control stack, and g identifies the current CFG. We also write $\triangleright_g$ for the abstract transition relation for a specific CFG g. The control stack for the abstract machine is constructed analogously to the control stack for the concrete machine. We write $\twoheadrightarrow$ for the augmented abstract transition relation. The rules for abstract calls and returns are as follows:

$$\frac{(S,Q,p) \triangleright_g (S',Q',p')}{((S,Q,p),A,g) \twoheadrightarrow ((S',Q',p'),A,g)}$$

$$\frac{g(p) = \mathsf{call}\ f, \quad f = (g',(a_1,\ldots,a_n)), \quad S = V_1 : \ldots : V_n : S'}{((S,Q,p),A,g) \twoheadrightarrow (([],Q\prod_i I_{a_i} + \sum_i V_i J_{a_i}, \mathsf{entry}\ f),(g,n(p),S') : A, g')}$$

$$\frac{g(p) = \mathsf{return}}{((V : S,Q,p),(g',p',S') : A,g) \twoheadrightarrow ((S',QI_{l_{\mathrm{R}}} + VJ_{l_{\mathrm{R}}},p'),A,g')}$$

Before we can state the soundness theorem for the inter-procedural analysis, we first need to define the notion of a safe model of a control stack.

Definition 5.22 Let $C = (g_1,p_1,c_1) : (g_2,p_2,c_2) : \ldots$ be a control stack, and $A = (g'_1,p'_1,S_1) : (g'_2,p'_2,S_2) : \ldots$ be an abstract control stack. Let m be a memory. The abstract control stack *models* the concrete control stack, written $A \Vdash m, C$, whenever:

1. $|A| = |C|$, and
2. $g_i = g'_i$, $p_i = p'_i$, and $S_i \models m, c_i$ for all stack indices i.

□

We can now formulate the soundness theorem extended to handle calls and returns.

Theorem 5.23 *Let $Q = (s, A_Q, t, u)$. If*

$$\begin{array}{rcl}
((c,m',p),C,g) & \Rightarrow & ((c',m'',p'),C',g') \\
((S,Q,p),A,g) & \twoheadrightarrow & ((S',Q',p'),A',g') \\
A & \Vdash & m, C \\
S & \models & m, c \\
m' & \sqsubseteq & \mathrm{img}(m,Q) \\
mt & \sqsubseteq & \mathsf{tr} \\
u & \sqsubseteq & \mathsf{tr}
\end{array}$$

then

$$\begin{array}{rcl} m'' & \sqsubseteq & \mathrm{img}(m, Q') \\ A' & \Vdash & m, C \\ S' & \models & m, c \end{array}$$

Proof. By cases on the definition of $\Rightarrow$. The first case follows directly from Theorem 5.18. The call case follows from Lemma 5.14 as the locations $a_1, \ldots, a_n$ are distinct; and the return case follows from Lemma 5.13 □

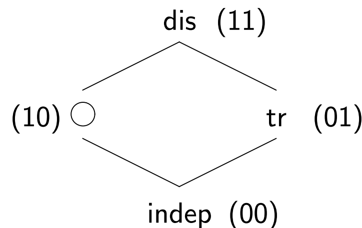
To get a terminating analysis of recursive functions we employ Theorem 5.21, together with taking the join over all paths, just as for intra-procedural loops.

Calls via function pointers are not currently treated. With a good "points to" analysis one can analyze a call via a pointer as the l.u.b. of calls to all the functions that the pointer might point to. This is analogous to control-flow analysis [Jon81b, Shi91].

The iterative implementation of the analysis is very close to the formalization given here. The implementation simulates the transitions of the abstract reduction system, and iteratively computes the Q component of the abstract configurations.

5.11 Practical Efficiency

The trust lattice can be embedded in the distributive lattice generated by two points as below (a boolean algebra), and the embedded lattice is closed under the operations of $\sqcup$ and $\sqcap$ on the four-point lattice:



The four-point lattice above can be implemented as two bits, using bitwise and and bitwise or for $\sqcap$ and $\sqcup$, respectively. This means that we can represent a vector of 16 trust-values in a single 32 bit machine word, and that we can perform 16 least upper bound and greatest lower bound operations in one instruction. This provides both low memory consumption and fast operations.

Sparse representations of the matrices involved in the trust analysis might be useful, as the matrices typically are sparse. On the other hand, the number of variables used by a procedure is usually very limited, so it might not pay off to use sparse representations. In any case: for the present implementation, the sparse matrix representations were deemed too complicated to implement for this purpose, and the lack of experience with dependence-based trust analysis also rendered the choice of a particular sparse representation difficult. However, the implementation of matrices chosen is suitably encapsulated in a class such that a re-implementation of that class would suffice to make the analyzer use a sparse representation of matrices.

The speed of the implementation is reasonably good; parsing and analyzing a particular 470 line C program takes 2 seconds on a SPARCServer 1000 running Solaris 2.5. This includes

running the compiler front-end written in Perl 5, the C pre-processor, and the trust analyzer. Adding the real compilation phase to this increases the time to 3.5 seconds, so the analysis is still rather costly compared to a "bare" compilation. If the analysis was integrated into a compiler a good amount of overhead could be saved. With the current implementation, a full compile pre-processes and parses the source code twice.

The speed of the analysis depends more on the number of variables used in the program than on the size of the code, as the size of the quads is quadratic in the number of variables, and quad multiplication is cubic in the number of variables. However, most programs use only relatively few variables in each procedure, and few global variables. In the implementation, the size of the quads used in the analysis of a procedure is bounded by the number of local variables in that procedure plus the number of global variables: each procedure has its own quad-size.

The table below exhibits a few data points about the performance of the analysis. The program "`851size2.c`" is an artificially constructed program written to test the analyzer on programs of differing size. The programs "`su.c`" and "`newgrp.c`" are implementations of the corresponding Unix commands. The "Variables" column lists the maximal number of variables visible at any point in the program.

File	Lines	Variables	Time (secs.)
`851size2.c`	474	14	1.9
`su.c`	112	22	2.1
`newgrp.c`	95	18	1.5

5.12 Discussion

When starting to program the analyzer it was hoped that it would be feasible to use it on existing large programs such as `sendmail` or `login`, however, it turned out that taking existing code and subjecting it to trust analysis is a pretty involved process. Detailed knowledge is needed about where untrusted input arrives, where appropriate checks are made, and where the dangerous operations are located. Unless given good documentation or having written the program by oneself this knowledge it hard to come by.

Also, a program designed without trust analysis in mind may be very hard to turn into a program with `check` and `trust` operations that the analysis will accept. A similar phenomenon exists when going from an untyped language to a typed language. A LISP program may be entirely safe and may never see a run-time error, but subjecting it to ML-like type inference may well fail.

In summary, to gain experience with the usefulness of trust analysis, new programs have to be written with the trust analysis in mind, just as new programs are written for ML with the ML type system in mind. This may also lead to the design of cleaner programs, as more discipline is enforced.

Another shortcoming of the current analysis is the overly conservative treatment of pointers. Pointers are used excessively in C due to the lack of, for example, a proper string type, but due to the lack of a good points-to analysis the current trust analysis insists that all pointers and things that may be pointed to are explicitly typed either trusted or untrusted. This may be viewed as no harder restriction than the existing type system for C, where all variables have to be declared with a certain type. But it is less flexible than the treatment of

simple variables, where the analysis is able to detect that the contents of a variable is trusted in some parts of the program, and untrusted in other parts.

In comparison with the trust analysis of Chapter 2, based on abstract interpretation and constraint solving, the analysis presented here builds on the dependence algebra technology. One advantage is that the analysis of C is program point sensitive, a variable need not have the same trustworthiness throughout its lifetime, as was the case in both analyses of Chapter 2. This allows for a kind of first-order polymorphism. Another advantage of the analysis presented here is that it directly deals with unstructured control flow. On the other hand, the analyses of Chapter 2 incorporate a simple pointer analysis which is lacking here.

The trust analysis of this chapter can easily be extended from the lattice of two trust-values to arbitrary trust lattices by the general theory of dependence quads developed in Chapter 4.

The trust analysis described in this chapter is about approximating the trustworthiness of *variables* and their run-time *values*. It is not aimed at guaranteeing the secure execution of dangerous *commands*. Consider the example (by T. Mogensen):

```
validated = FALSE;
while (!validated) {
  // read password and set validated to TRUE if OK.
}
// perform dangerous command
```

Suppose, because of some programming error, that `validated` may depend on some untrustworthy variables (and is therefore classified `untrusted`), and that the loop may therefore terminate on false premises. Then the dangerous command will be executed on false premises without any complaints from the analysis, as the dangerous command is not control dependent on the untrustworthy variable `validated`.

A method to handle this problem is to regard the program counter as a variable as well. The program counter starts out as trustworthy, pointing to the entry point of the program, but is classified a untrustworthy as soon as it is changed under untrusted control. This approach has also been used in the field of security flow analysis.

5.13 Summary

This chapter described the implementation of a trust analyzer for the C programming language, utilizing the dependence quadruple technology developed in Chapter 4. As such it presented a practical application of the dependence algebra theory to a concrete program analysis problem, and it connected the subjects of dependence analysis and trust analysis, the two main themes of the present thesis.

The theory of control dependence was utilized to handle unstructured control flow. A conservative evaluation order dependence analysis of C has also been developed and implemented in order to help the other analyses.

We have given formal definitions of (an appropriately abstract) concrete semantics for the stack machine language used internally in the analyzer, and of the analysis carried out, and the soundness of the analysis is proved (Theorem 5.23). The separability of the analysis has also been proved (Theorem 5.21).

Chapter 6

Soft Type Inference for Action Semantic Equations

Action may not always bring happiness;
but there is no happiness without action.
B. Disraeli

6.1 Motivation

Developing a formal semantics for a programming language can be a formidable task. The action semantics formalism [Mos92] strives to ease the development by providing intuitive and readable constructions for many common features of programming languages. Still, making sure that a semantics is consistent and works as expected, is difficult.

Type inference and checking is used in many programming languages to find simple errors at an early stage. This chapter begins to develop a soft type inference scheme for a subset of action notation. The type inference is intended to be run on the equations describing the dynamic semantics of a language, that is, it is intended to be run before a particular program has been submitted to the semantics for interpretation.

There are two things that make type inference at the level of action semantic equations difficult. First, the absence of a particular program: only small holed fragments of actions are present on the right hand sides of the equations. Second, the use of flat heterogeneous tuples to pass data around among primitive actions. Consider for example the action:

give (1,2,true) and give (false,5)

The action combinator "and" concatenates the data tuples *given* by the two sub-actions, and the combined action is said to *give* the tuple $(1, 2, \mathsf{true}, \mathsf{false}, 5)$.

Individual components of tuples can be extracted by number and tuples can be concatenated. Furthermore, sort intersection checks can be performed on arbitrary tuples and components. The following action shows the use of sort intersection for run-time typechecking:

A_1 then give the given truth-value#4 .

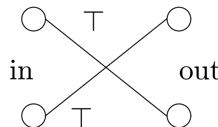
The action combinator "then" corresponds to functional composition, taking the tuple given by the first sub-action (A_1) and handing it over to the second sub-action. The *yielder* "the given truth-value#4" selects the fourth component of the given tuple and intersects it with the sort truth-value. If the action A_1 is instantiated to be the action of the previous example, the combined action would then give the truth-value false. However, if the fourth component of the tuple given by A_1 does not intersect the sort truth-value (or the tuple does not have a fourth component at all), the combined action is said to *fail*, that is, terminate abnormally.

The above operations make it necessary for a type system to keep track of the tuples at a very fine-grained level. The dependence algebra technology of Chapter 4 was developed primarily for this purpose.

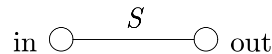
Our type inference algorithm is "soft" in the sense that it is forgiving: no sets of equations are forbidden as not being type correct. This means that the type system does not limit the set of allowed actions, and that it may infer incomplete (overly conservative) information in some cases.

The action combinators are polymorphic: the and combinator concatenates arbitrary tuples. The sorts (types) used in action notation are ordered in a lattice and sub-sort checks are specified in terms of sort intersection. These features call for a type system that is able to handle both polymorphism and sub-typing together with the extra complication of tuple concatenation. Type-inference for systems combining sub-typing with polymorphism and higher-order languages has proven to be quite hard, and only recently some inroads have been made [AW93, EST95, OL96]. We eschew this hard problem by integrating a control flow analysis (along the lines of Shivers's 0-CFA [Shi91]) into the analysis.

Parametric polymorphism allows recording of dependences between inputs and outputs. In the polymorphic type: $(\alpha, \beta) \to (\beta, \alpha)$ we can see that the first argument to the function ends up in the second position in the output, and vice versa. In our matrix-based setup this is modeled by having edges in the dependence quad connecting the first input component to the second output component and vice versa.



The type of a function doing sort intersection could look like: $\alpha \to (\alpha \wedge S)$ showing that the function accepts any argument, but applies a sort intersection with S to it. In our setup this is modeled by the dependence algebra elements on the edges of our matrices, so for this function there would be an edge between the input component and the output component labeled with the representation of S, as pictured below:



If only forwards type information was desired, i.e., information about the output type of actions under certain conditions, then a relatively simple abstract interpretation analysis akin to the type analysis in [Ørb93, Ørb94a] would suffice. However, we also want information about the demands that the action places on the input.

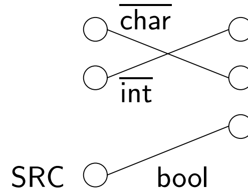
This kind of backwards type information can be obtained from the dependence quadruple via the reverse image operation. One could imagine a more traditional backwards abstract interpretation analysis to approximate demands on input, but due to the heavy use of tuples and their flat concatenation, this would quickly loose precision compared to the quad-based

approach.

Example 6 As an example of the flavor of the analysis consider the action:

| give the given integer#2 and give the given character#1
then regive and give true .

The analysis would essentially compute the quad illustrated below for the above action. The overlined types stem from the must/may construction of Section 4.10.



The above statement is not entirely correct. The analysis would associate a whole set of quads of the above form with the action, the quads only differing in the length of the input they accept. The output type of the action can be found using the image operation: $\mathrm{img}(\top, Q)$, and the input type can be found as $\mathrm{rev}(Q, \top)$. ◇

6.1.1 Related Work

Action notation does not have records per se, but it has been suggested that one could re-use some of the record typing machinery for a type system for action notation. Record typing schemes for ML and related languages have been studied by Wand [Wan87, Wan88, Wan89], Rémy [Rém89, Rém92] and others with the aim of typing object oriented languages. These systems treat different kinds of record concatenation and extension. A record type is a finite map from *labels* to types, such as {a:int, b:boolean}. Concatenating the records {a:int} and {b:boolean} yields the record {a:int, b:boolean}. However, standard action notation does *not* have records in the functional facet: it has flat tuples. We can encode the 1-tuple (int) as a record: {1:int}. Using this idea together with tuple concatenation we get

$$\{1 : \mathsf{int}\} +\!\!\!+ \{1 : \mathsf{boolean}\} = \{1 : \mathsf{int}, 2 : \mathsf{boolean}\}.$$

Note that the label of the boolean type *changes*, showing that tuple concatenation is not record concatenation.

The label-selective λ-calculus by Garrigue and Aït-Kaci [GAK94, AKG95] allows for records with numerical labels and a variant of record concatenation that can be used to model tuple concatenation. However, modeling the action combinator "and" in that calculus would require a whole set of terms (a term for each input length), and type inference for that calculus has not been investigated much.

Type inference for actions has previously been explored by Even and Schmidt [ES90b, ES90a] in a categorical framework for an earlier version of action notation. There are two main differences between their work and the work presented here: their version of action notation does not use flat tuples to pass around data, instead data is packed in records with labels for individual components. The use of flat tuples as is now used in action notation presents different problems that cannot be handled by the record typing technology employed by Even and Schmidt. The second difference between our system and theirs is that their type system is "hard", an action can be found type incorrect, and no approximate type is found.

The ASTIS system of Jakobsen [Jak93] was a practical implementation and extension of the techniques developed by Even and Schmidt. The Actress compiler generator by Brown, Moura and Watt [BMW92, MW94] also applied an extended version of the type system by Even and Schmidt.

6.2 Introduction to Action Semantics

The following introduction to action semantics is based on the introduction given in [Ørb94a].

Action semantics is a formalism for the description of the dynamic semantics of programming languages, developed by Mosses and Watt [Mos92]. Based on an order-sorted algebraic framework, an action semantic description of a programming language specifies a translation from abstract terms of the source language to action notation.

Action notation is designed to allow comprehensible and accessible semantic descriptions of programming languages; readability and modularity are emphasized over conciseness. Action semantic descriptions scale up well, and considerable reuse of descriptions is possible among related languages. An informal introduction to action notation, as well as the formal semantics of the notation, can be found in [Mos92].

The semantics of action notation is itself defined by a structural operational semantics, and actions reflect the gradual, stepwise, execution of programs. The performance of an action can terminate in one of three ways: It may *complete*, indicating normal termination; it may *fail*, to indicate the abortion of the current alternative; or it may *escape*, corresponding to exceptional termination which may trapped. Finally, the performance of an action may *diverge*, ie. end up in an infinite loop.

Actions may be classified according to which *facet* of action notation they belong. There are five facets:

- the *basic* facet, dealing with control flow regardless of data.
- the *functional* facet, processing *transient* information, actions are *given* and *give* data.
- the *declarative* facet, dealing with bindings (*scoped information*), actions *receive* and *produce* bindings.
- the *imperative* facet, dealing with loads and stores in memory (*stable information*), actions may *reserve* and *unreserve* cells of the storage, and change the contents of the cells.
- the *communicative* facet, processing *permanent information*, actions may *send* and *receive* messages communicated between processes.

In general, imperative and communicative actions actions are *committing*, which prevents backtracking to alternative actions on failure. There are also hybrid actions that deal with more than one facet. Below are some example action constructs:

- 'complete': the simplest action. Unconditionally completes, gives no data and produces no bindings. Not committing.

- 'A_1 and A_2': a basic action construct. Each sub-action is given the same data as the combined action, and each receives the same bindings as the combined construct. The data given by the two sub-actions is concatenated into a tuple to form the data given by the combined action, (the construct is said to be *functionally conducting*). The performance of the two sub-actions may be interleaved.

- 'A_1 or A_2': a basic action construct, represents non-deterministic choice between the two sub-actions. Either A_1 or A_2 is performed. If A_1 fails without committing A_2 is performed, and vice versa.

- 'store Y_1 in Y_2': an imperative action. Evaluates the yielder Y_1 and stores the result in the cell yielded by Y_2. Commits and completes when Y_1 evaluates to a storable and Y_2 evaluates to a cell.

An action term consists of constructs from two syntactic categories, there are action constructs like those described above, and there are yielders that we will describe below. Yielders may be evaluated in one step to yield a value. Below are a few example yielders:

- 'sum(Y_1, Y_2)': evaluates the yielders Y_1 and Y_2 and forms the sum of the two numbers.

- 'the given $D\#n$': picks out the n'th element of the tuple of data given to the containing action. Yields the empty sort nothing unless the n'th item of the given data is of sort D.

- 'the D stored in Y': provided that Y yields a cell, it yields the intersection of the contents of that cell and the sort D.

The data transfered between actions is collected into *sorts*, which are like sets of data except that singular items of data are considered sorts of one element. The sorts form a distributive lattice under the sort inclusion order ($\leq$). Sort union is written as a vertical bar (|), and sort intersection is written with the ampersand symbol (&). The empty sort nothing denotes the bottom of the lattice.

As an example we give below an action semantics for a call-by-value λ-calculus with constants.

6.2.1 Abstract Syntax

needs: Numbers/Integers(integer), **Strings**(string) .

grammar:

(1) Expr = ⟦ "lambda" Var "." Expr ⟧ | ⟦ Expr "(" Expr ")" ⟧ |
⟦ Expr "+" Expr ⟧ | integer | Var .

(2) Var = string .

6.2.2 Semantic Functions

includes: Abstract Syntax .

introduces: evaluate _ .

- evaluate _ :: Expr → action .

(1) evaluate I:integer = give I .

(2) evaluate V:Var = give the datum bound to V .

(3) evaluate ⟦ "lambda" V:Var "." E:Expr ⟧ =
give the closure abstraction of
| furthermore bind V to the given datum#1
| hence evaluate E .

(4) evaluate ⟦ E_1:Expr "(" E_2:Expr ")" ⟧ =
| evaluate E_1 and evaluate E_2
then enact application the given abstraction#1 to the given datum#2 .

(5) evaluate ⟦ E_1:Expr "+" E_2:Expr ⟧ =
| evaluate E_1 and evaluate E_2
then give the sum of them .

6.2.3 Semantic Entities

includes: Action Notation .

- datum = abstraction | integer | □ .
- bindable = datum .
- token = string .

Note the different use of semantic brackets (⟦·⟧) in action notation and elsewhere. In action notation the semantic brackets are defined as tree constructors, whereas they are ordinarily associated with the semantic functions themselves.

6.3 A Subset of Action Notation

Here we give the syntax of the subset of action notation that we treat in this chapter. The subset merely consists of the essential parts of the functional facet.

This particular subset of action notation is chosen to facilitate the investigation of the problems with data passing among actions in the form of flat, heterogeneous tuples. Actions from the declarative and imperative facets can easily be accommodated as consumers and producers of transient data using the src and sink parts of the dependence quads of Chapter 4. Moreover, the reason for not incorporating those facets in the subset considered here is that bound tokens and locations in the store are most often not known until a particular source program is given. As the analysis developed in this chapter works at the level of semantic equations, such concrete tokens and locations are not present at the time of analysis. Another simple way to accommodate imperative actions would be to consider all locations equal and have one slot representing all locations in the store, however, this would also be excessively imprecise.

As to the communicative facet, it is outside the scope of this chapter. If the communicating agents can be distinguished at the level of the semantic equations they can be analyzed

independently, but this might seldom be the case. In more complicated cases the control flow analysis would have to be extended to handle communicative actions correctly.

In the following we define the abstract syntax of our subset of action notation. The grammar is written in the ordinary BNF style instead of in the special action notation style. The action notation style will be used only for examples of actions and action semantic specifications, not at the meta-level.

$$
\begin{array}{rcl}
\textit{Action} & ::= & \textit{Primitive} \mid \textit{Composite} \\
\textit{Primitive} & ::= & \mathsf{complete} \mid \mathsf{fail} \mid \mathsf{escape} \mid \mathsf{unfold} \\
\textit{Composite} & ::= & \textit{Action}\ \mathsf{and}\ \textit{Action} \mid \textit{Action}\ \mathsf{or}\ \textit{Action} \\
 & & \mid \textit{Action}\ \mathsf{then}\ \textit{Action} \mid \textit{Action}\ \mathsf{trap}\ \textit{Action}\mathsf{give}\ \textit{Yielder} \mid \mathsf{check}\ \textit{Yielder} \\
 & & \mid \mathsf{unfolding}\ \textit{Action} \mid \mathsf{enact\ application}\ \textit{Yielder}\ \mathsf{to}\ \textit{Yielder} \\
\textit{Yielder} & ::= & \mathsf{it} \mid \mathsf{them} \mid \mathsf{sum}\ \textit{Yielder} \mid \mathsf{the\ given}\ \textit{Sort} \\
 & & \mid \mathsf{the\ given}\ \textit{Sort}\ \#\ \textit{Natural} \mid \textit{Yielder}\ \mathsf{is}\ \textit{Yielder} \\
 & & \mid \mathsf{abstraction\ of}\ \textit{Action} \mid \textit{Natural} \mid \textit{Boolean} \\
 & & \mid (\textit{Yielder}, \textit{Yielder}) \\
\textit{Natural} & ::= & 1 \mid 2 \mid 3 \mid \ldots \\
\textit{Boolean} & ::= & \mathsf{true} \mid \mathsf{false} \\
\textit{Sort} & ::= & \mathsf{integer} \mid \mathsf{truth\text{-}value} \mid \mathsf{abstraction} \mid \mathsf{nothing} \mid \textit{Sort}\ \text{“}|\text{”}\ \textit{Sort} \\
 & & \mid \textit{Sort}\ \&\ \textit{Sort} \mid (\textit{Sort}, \textit{Sort})
\end{array}
$$

6.4 A Natural Semantics

This section defines a natural semantics for our subset of action notation. We have chosen to give a natural (big step) operational semantics for our subset of action notation, both to make the chapter self-contained, and to avoid the complexities of the full operational semantics of action notation given in [Mos92]. Similar approaches have been used in papers by others on action semantics, most notably by Palsberg in his thesis [Pal92]. The semantics given here is basically a subset of the semantics of Palsberg's thesis.

Simple values in the operational semantics are integers, booleans, and abstraction closures. Heterogeneous tuples of simple values form the data values of interest.

We define the operational semantics as an evaluation relation between configurations and results. A configuration (A, B, v) consists of an action term (A) to be evaluated, the body of the innermost unfolding enclosing A (B), and a tuple of data given to the action (v). A result (t, w) consists of a tag (t) signifying whether the action completed, failed, or escaped together with a tuple of data (w) given by (or escaping from) the action. The evaluation relation is written:

$$(A, B, v) \to (t, w).$$

The evaluation of yielders is specified in the same manner as the evaluation of actions. A yielder configuration (Y, B, v) (where Y is the yielder, B is the innermost enclosing unfolding, and v is the tuple of data given to the yielder) evaluates to a tuple w, written:

$$(Y, B, v) \to w.$$

Abstractions may contain free unfolds that are lexically scoped. The yielder evaluation must therefore keep track of the innermost enclosing unfolding just like the evaluation of actions. This is a conservative extension of standard action notation proposed by Lassen [Las95, Las97]. We write $\langle A, B\rangle$ for the abstraction with the body A and enclosing unfolding B. Notice that the unfolding is packaged up together with the body in the creation of the abstraction to ensure lexical scoping of unfoldings.

We first define a concrete interpretation of sort expressions. The function σ associates the syntax of a sort with its denotation. The sort lattice used in the concrete semantics is the distributive sort-inclusion lattice generated from the atoms: integer, boolean, abstraction, and tuples of these sorts. The empty sort nothing denotes the bottom of the lattice. Tuples are ordered componentwise, and tuples of different lengths are not related. Tuples with one component are identified with singular data items.

$$
\begin{array}{rcl@{\qquad}rcl}
\sigma[\![\mathsf{integer}]\!] & = & \mathsf{integer} & \sigma[\![\mathsf{truth\text{-}value}]\!] & = & \mathsf{boolean} \\
\sigma[\![\mathsf{abstraction}]\!] & = & \mathsf{abstraction} & \sigma[\![(S_1, S_2)]\!] & = & \sigma[\![S_1]\!] +\!\!\!+ \, \sigma[\![S_2]\!] \\
\sigma[\![S_1 \mid S_2]\!] & = & \sigma[\![S_1]\!] \sqcup \sigma[\![S_2]\!] & \sigma[\![S_1 \;\&\; S_2]\!] & = & \sigma[\![S_1]\!] \sqcap \sigma[\![S_2]\!] \\
\sigma[\![\mathsf{nothing}]\!] & = & \mathsf{nothing} & & &
\end{array}
$$

The sort integer contains all the integers, and the sort boolean contains the two individuals true and false. The sort abstraction contains all closures. We write $\sqsubseteq$ for sort inclusion, and $\sqcup$ and $\sqcap$ for least upper bound and greatest lower bound, respectively.

6.4.1 Evaluation of Actions

The evaluation relations are defined as sets of inference rules. A configuration evaluates to a result only if it can be derived from these rules.

$$\frac{}{(\mathsf{complete}, B, v) \to (\mathsf{completed}, ())} \qquad \frac{}{(\mathsf{escape}, B, v) \to (\mathsf{escaped}, v)}$$

$$\frac{}{(\mathsf{fail}, B, v) \to (\mathsf{failed}, ())} \qquad \frac{(B, B, v) \to (t, w)}{(\mathsf{unfold}, B, v) \to (t, w)}$$

$$\frac{(A, \mathsf{unfolding}\ A, v) \to (t, w)}{(\mathsf{unfolding}\ A, B, v) \to (t, w)}$$

$$\frac{(A_1, B, v) \to (\mathsf{completed}, w_1) \quad (A_2, B, v) \to (\mathsf{completed}, w_2)}{(A_1 \text{ and } A_2, B, v) \to (\mathsf{completed}, w_1 +\!\!+ w_2)}$$

$$\frac{(A_1, B, v) \to (t, w_1) \quad t \in \{\mathsf{escaped}, \mathsf{failed}\}}{(A_1 \text{ and } A_2, B, v) \to (t, w_1)}$$

$$\frac{(A_2, B, v) \to (t, w_2) \quad t \in \{\mathsf{escaped}, \mathsf{failed}\}}{(A_1 \text{ and } A_2, B, v) \to (t, w_2)}$$

$$\frac{(A_1, B, v) \to (t, w_1) \quad t \in \{\mathsf{completed}, \mathsf{escaped}\}}{(A_1 \text{ or } A_2, B, v) \to (t, w_1)}$$

$$\frac{(A_2, B, v) \to (t, w_2) \quad t \in \{\mathsf{completed}, \mathsf{escaped}\}}{(A_1 \text{ or } A_2, B, v) \to (t, w_2)}$$

$$\frac{(A_1, B, v) \to (\mathsf{failed}, w_1) \quad (A_2, B, v) \to (\mathsf{failed}, w_2)}{(A_1 \text{ or } A_2, B, v) \to (\mathsf{failed}, ())}$$

$$\frac{(A_1, B, v) \to (\mathsf{completed}, w_1) \quad (A_2, B, w_1) \to (t, w_2)}{(A_1 \text{ then } A_2, B, v) \to (t, w_2)}$$

$$\frac{(A_1, B, v) \to (t, w_1) \quad t \in \{\mathsf{escaped}, \mathsf{failed}\}}{(A_1 \text{ then } A_2, B, v) \to (t, w_1)}$$

$$\frac{(A_1, B, v) \to (t, w_1) \quad t \in \{\mathsf{completed}, \mathsf{failed}\}}{(A_1 \text{ trap } A_2, B, v) \to (t, w_1)}$$

$$\frac{(A_1, B, v) \to (\mathsf{escaped}, w_1) \quad (A_2, B, w_1) \to (t, w_2)}{(A_1 \text{ trap } A_2, B, v) \to (t, w_2)}$$

$$\frac{(Y, B, v) \to w \neq \mathsf{nothing}}{(\text{give } Y, B, v) \to (\mathsf{completed}, w)} \qquad \frac{(Y, B, v) \to \mathsf{nothing}}{(\text{give } Y, B, v) \to (\mathsf{failed}, ())}$$

$$\frac{(Y, B, v) \to \mathsf{true}}{(\text{check } Y, B, v) \to (\mathsf{completed}, ())} \qquad \frac{(Y, B, v) \to w \neq \mathsf{true}}{(\text{check } Y, B, v) \to (\mathsf{failed}, ())}$$

$$\frac{(Y_1, B, v) \to \langle A', B' \rangle \quad (Y_2, B, v) \to w_2 \quad (A', B', w_2) \to (t, w_3)}{(\text{enact application } Y_1 \text{ to } Y_2, B, v) \to (t, w_3)}$$

6.4.2 Evaluation of Yielders

The following rules specify the yielder evaluation relation. In addition to the rules below we take the meta-rule that whenever no other rule applies to (Y, B, v), it evaluates to nothing. Note that we for simplicity allow 'is' to compare arbitrary data.

$$\frac{|v| = 1}{(\mathsf{it}, B, v) \to v} \qquad \frac{v \sqsubseteq \sigma[\![S]\!]}{(\mathsf{the\ given}\ S, B, v) \to v}$$

$$\frac{}{(\mathsf{them}, B, v) \to v} \qquad \frac{|v| \geq n \quad v_n \sqsubseteq \sigma[\![S]\!]}{(\mathsf{the\ given}\ S\ \#\ n, B, v) \to v_n}$$

$$\frac{|v| \neq 1}{(\mathsf{it}, B, v) \to \mathsf{nothing}} \qquad \frac{(Y_1, B, v) \to w \quad (Y_2, B, v) \to w}{(Y_1\ \mathsf{is}\ Y_2, B, v) \to \mathsf{true}}$$

$$\frac{}{(b, B, v) \to b} \qquad \frac{(Y_1, B, v) \to w_1 \quad (Y_2, B, v) \to w_2 \quad w_1 \neq w_2}{(Y_1\ \mathsf{is}\ Y_2, B, v) \to \mathsf{false}}$$

$$\frac{}{(n, B, v) \to n} \qquad \frac{(Y_1, B, v) \to w_1 \quad (Y_2, B, v) \to w_2}{((Y_1, Y_2), B, v) \to w_1 +\!\!+ w_2}$$

$$\frac{(Y, B, v) \to w \quad w_i \sqsubseteq \mathsf{integer}}{(\mathsf{sum}\ Y, B, v) \to \sum_i w_i} \qquad \frac{}{(\mathsf{abstraction\ of}\ A, B, v) \to \langle A, B \rangle}$$

Notice the checks for subsort inclusion in the two yielders: "the given" and "the given #". In full action semantics the subsort checks are defined via sort intersection instead of subsort inclusion. We design our semantics with sort inclusion because we want to be able to obtain a stronger soundness theorem than would otherwise be obtainable. With the current definition we get an analysis that is able to say that the type of the actual output *is less than* the type computed by the analysis, and similarly for the input. With the official action semantic definitions a soundness theorem would only be able to state the weaker property that the type of the output *has a non-empty intersection* with the type computed by the analysis. This property is not sufficiently strong to support an inductive soundness proof. The basic problem is that even if we know that:

$$s \wedge f(t') \neq \bot \text{ and } t' \wedge g(t) \neq \bot,$$

we cannot conclude that:

$$s \wedge f(g(t)) \neq \bot.$$

In the common case where run-time data are not proper sorts, the two kinds of soundness results coincide, because if a is an atom in a lattice (corresponding to a singleton sort) then $a \wedge b \neq \bot$ if and only if $a \leq b$.

6.5 Simplification

The analysis of a set of semantic equations consists of three phases. First the equations are simplified to eliminate unfold and unfolding, then they are transformed into a set of equations in a simpler language, and finally the new set of equations is analyzed.

This section describes the simplification of action semantic equations. There are three goals of the simplification phase:

1. replace unfold nodes by applications of (new) semantic functions and thereby also eliminate the need for the unfolding construct. This essentially serves to give a name to each unfolding.
2. associate a unique semantic function with the body of each syntactic abstraction in the semantics. The names generated for abstractions in this manner are used in the control flow part of the ensuing analysis.
3. add new semantic functions such that each semantic function has only one defining equation.

As an example of the first simplification, consider the equation:

(1) foo⟦"bar"⟧ = unfolding complete then unfold .

This is simplified into the equations (foo' is a new function):

(1) foo⟦"bar"⟧ = foo'⟦"bar"⟧ .

(2) foo'⟦"bar"⟧ = complete then foo'⟦"bar"⟧ .

Notice that as a result of this simplification, the semantics becomes non-compositional, and here foo⟦"bar"⟧ expands to an infinite action instead of a finite one. The simplification essentially replaces one finite representation of an infinite action (unfolding, unfold) with another: the equational representation. The simplified equations are not intended to be used for expanding a concrete program into its corresponding action, they merely serve as an intermediate representation suitable for the ensuing analysis.

The second simplification is illustrated by the following equation, where baz is some semantic function:

(1) foo⟦"bar" X⟧ = baz X then give abstraction of give 7 .

which is simplified into:

(1) foo⟦"bar" X⟧ = baz X then give abstraction of foo'⟦"bar" X⟧ .

(2) foo'⟦"bar" X⟧ = give 7 .

In effect the simplification ensures that the body of an abstraction is the application of a unique semantic function, which can be used to identify the syntactic abstraction. This corresponds to assigning labels to λ's in control flow analysis of functional languages.

The final part of the simplification phase is illustrated by:

(1) foo⟦"bar"⟧ = give 8 .

(2) foo⟦"baz"⟧ = give 10 .

which is transformed into:

(1) foo E = foo'⟦"bar"⟧ or foo"⟦"baz"⟧ .

(2) foo'⟦"bar"⟧ = give 8 .

(3) foo"⟦"baz"⟧ = give 10 .

where the function foo' and foo" are new. In the absence of a concrete program the last set of equations gives a sound approximation of the data flow in the actions generated by the function foo of the original equations. This hinges on the fact that no data is transferred when an action fails, as opposed to when it completes or escapes.

6.5.1 Simplification Algorithm

The simplification algorithm works in three phases. First additional equations for unfoldings and abstractions are added, then a set of enclosing unfoldings for each occurrence of unfold is computed, and finally each occurrence of unfold is replaced by the or of a set of non-terminals.

Let NT be the set of *non-terminals*, i.e., the set of applications of semantic functions, such that the argument of the function is ignored. In other words: the applications foo⟦"baz"⟧ and foo⟦"quux" X⟧ denote the same non-terminal, whereas bar⟦"baz"⟧ denotes a different non-terminal. The non-terminals may be underlined, or non-underlined. Underlines are used to distinguish non-terminals associated with unfoldings.

Phase one consists of applying the following rules to the right hand side of each equation. Writing $A \to (A', E')$ means that the action term A simplifies to the term A' with the addition of the extra equations E'. The first three rules are to be taken as meta-rules that show how the relation is defined for all action- and yielder-constructs except for the special cases explicitly mentioned in the rules below. For unfoldings and abstractions we generate fresh non-terminals, this is what the notation "N fresh" means.

$$\frac{}{\mathit{Primitive} \to (\mathit{Primitive}, \emptyset)} \tag{6.1}$$

$$\frac{A_1 \to (A_1', E_1)}{\mathit{Unary}\ A_1 \to (\mathit{Unary}\ A_1', E_1)} \tag{6.2}$$

$$\frac{A_1 \to (A_1', E_1) \quad A_2 \to (A_2', E_2)}{A_1\ \mathit{Binary}\ A_2 \to (A_1'\ \mathit{Binary}\ A_2', E_1 \cup E_2)} \tag{6.3}$$

$$\frac{A \to (A', E') \quad N \text{ fresh}}{\mathsf{unfolding}\ A \to (\underline{N}, E' \cup \{\underline{N} = A'\})} \tag{6.4}$$

$$\frac{}{N \to (N, \emptyset)} \tag{6.5}$$

$$\frac{A \to (A', E') \quad N \text{ fresh}}{\mathsf{abstraction\ of}\ A \to (\mathsf{abstraction\ of}\ N, E' \cup \{N = A'\})} \tag{6.6}$$

All the new equations map fresh non-terminals to action terms, so we can represent the set of equations as a simple linked list, and thus the first phase runs in $O(n)$ time where n is the size of the original semantics. After all the original equations are processed, duplicate definitions of the same non-terminal are transformed by adding new non-terminals and adding

an or action as described above. This can be done in $O(n \log(n))$ time by first sorting the list of equations with the left hand sides as keys.

In phase two we derive a directed graph from the set of equations produced in phase one. For each non-terminal we have a node in the graph, and for each occurrence of unfold in the semantics we also have a node. For each equation '$N = A$' we add edges from the node corresponding to N to the nodes corresponding to the non-terminals and unfolds in A. This graph G can be constructed in $O(n)$ time. From the graph G we form the opposite graph G^{op} by reversing all the edges.

For each node U corresponding to an occurrence of unfold we do a depth-first traversal (DFS) of G^{op} starting from U. During this traversal, whenever a node corresponding to a non-underlined non-terminal is met, the DFS just continues; whenever a node corresponding to an underlined non-terminal (stemming from an unfolding construct) is met, we add that node to the set $\text{Unf}(U)$, and the DFS does not continue beyond that node[1]. As the number of edges in the graph G^{op} is in $O(n)$ this takes at most $O(n^2)$ time. When the algorithm finishes, $\text{Unf}(U)$ contains the set of unfoldings possibly immediately enclosing the occurrence of unfold corresponding to U.

Phase three replaces each occurrence of unfold in the semantics with the or of the non-terminals in $\text{Unf}(U)$. In total the simplification can be done in quadratic time.

In order to reason about the semantics of actions with embedded non-terminals, so that we avoid expanding simplified equations to infinite actions, we extend the operational semantics with the following cases for non-terminals in actions and yielders. Let

$$S : NT \to \mathit{Action} \cup \mathit{Yielder} \cup NT$$

denote the (simplified) equations.

$$\frac{(S(N), B, v) \to (t, w)}{(N, B, v) \to (t, w)}$$

$$\frac{(S(N), B, v) \to w}{(N, B, v) \to w}$$

This semantics ignores arguments to semantic functions, as they are not present at the time of analysis. If P is a program, f a semantic function, $t \in \{\mathsf{completed}, \mathsf{escaped}\}$, and $(f(P), B, v) \to (t, w)$ according to the original semantics then also $(N, B, v) \to (t, w)$ is possible in the simplified equation system, where N is the non-terminal corresponding to f.

It is possible that where an action generated from the original semantics may fail, the action generated from the simplified semantics may not, due to the replacement of multiple definitions of the same semantic function with the or construct. This does not cause trouble as the or construct is symmetric, and no data is passed from failing actions. An analysis of the termination mode (completed, escaped or failed) of the actions generated by a semantic function may be built along the lines of the termination analysis in [Ørb94a, Ørb93].

6.6 The Dependence Algebra

This section presents the dependence algebra used in the analysis. The construction of the algebra starts from a distributive type lattice, an abstraction of the sub-sort lattice used by the

[1]The idea of this algorithm is due to Gerth Stølting Brodal.

action semantics that one is interested in analyzing. From the type lattice a **DAlg**-structure is then constructed via the must-may construction of Chapter 4.

To make the exposition more concrete we use the free distributive lattice generated from the flat meet-semilattice with the atoms: int, bool, and abs_N, where the atom abs_N corresponds to the abstraction identified by the non-terminal N in the simplified semantics.

The distributive lattice is constructed by first generating the smallest ring of sets (a family of sets closed under union and intersection) containing the singleton sets: {int}, {bool}, and $\{\mathsf{abs}_N\}$ for each N. This ring of sets is isomorphic to a distributive lattice (using the subset ordering) via the Birkhoff-Stone theorem. This distributive lattice satisfies the requirements of Theorem I.5.5 in [Grä78] proving the existence of the free distributive lattice generated from the flat meet-semilattice. This construction can be generalized to any finite flat meet-semilattice.

We write $(L, \leq, \vee, \wedge)$ for the lattice generated in this way. Notice that the lattice does not in itself contain tuples, they are handled by the quads built on top of L. We write $\perp$ for the bottom element of L, corresponding to the bottom of the concrete sort lattice: nothing.

From the lattice L we first construct the **DAlg**-structure $(\overline{L}_{\mathsf{indep}}, \sqcup, +, \cdot, \mathbf{0}, \mathbf{1})$ as in Section 4.10. The unit, $\mathbf{1}$, of the constructed algebra $\overline{L}_{\mathsf{indep}}$ is the overlined copy of the top of the underlying type lattice. The zero, $\mathbf{0}$, of the algebra is indep: the special element added to signify independence.

Define the function typeof which maps sorts to vectors of types of L. Sorts may contain the empty tuple, so the vectors returned by typeof are indexed starting from 0. We first define the function for individual values:

$$\begin{aligned} \mathsf{typeof}(n) &= (\perp, \mathsf{int}, \perp, \ldots) & \mathsf{typeof}(b) &= (\perp, \mathsf{bool}, \perp, \ldots) \\ \mathsf{typeof}(\langle N, B\rangle) &= (\perp, \textstyle\bigvee_N \mathsf{abs}_N, \perp, \ldots) & \mathsf{typeof}(\mathsf{nothing}) &= (\perp, \perp, \ldots). \end{aligned}$$

In effect, typeof() takes a sort and sorts its elements according to length. We define the set of lengths of a sort s by the following: Let v be a value, and n a natural number then:

$$v \sqsubseteq s \text{ and } |v| = n \Rightarrow n \in |s|.$$

For proper sorts we specify:

$$\begin{aligned} \mathsf{typeof}(s)_i &= \bigvee_{(s' \sqsubseteq s \;\mid\; i = |s'|)} s' \\ \mathsf{typeof}(s +\!\!+ s')_i &= \bigvee_{(m+n=i, m \in |s|, n \in |s'|)} \mathsf{typeof}(s)_m +\!\!+ \mathsf{typeof}(s')_n. \\ \mathsf{typeof}(s \sqcup s')_i &= \mathsf{typeof}(s)_i \vee \mathsf{typeof}(s')_i \\ \mathsf{typeof}(s \sqcap s')_i &= \mathsf{typeof}(s)_i \wedge \mathsf{typeof}(s')_i \;. \end{aligned}$$

It is not hard to see from the definition that typeof is monotone as stated below.

Proposition 6.1 *The function* typeof *from sorts to type vectors is monotone in each component: If $s \sqsubseteq s'$ then*

$$\mathsf{typeof}(s)_i \leq \mathsf{typeof}(s')_i \;.$$

6.7 Translation

The set of simplified action semantic equations are translated into a set of equations over a smaller language. This way yielders, as well as completing and escaping actions can be handled in a unified way, and the specification of the analysis becomes simpler.

Expressions in the new intermediate language are formed according to the following grammar:

$$E ::= N \mid T \mid EE \mid E +\!\!+ E \mid E \cup E \mid \lambda N \mid E @ E.$$

Expressions are non-terminals (applications of semantic functions, as before), terminals (T for leaves in the action tree), composition, concatenation, and union of expressions. Furthermore, expressions can be abstractions of non-terminals (due to the previous simplification), and function application.

The expressions of this language are to be interpreted over sets of dependence quadruples, so the terminals are specified as such sets.

The translation consists of three functions that map simplified action terms to terms over the new language. $X_{\mathsf{completed}}$ maps action terms to intermediate expressions that will model the completing behavior of the action term. X_{escaped} works analogously but for escaping behavior. X_{Y} maps yielder terms to the intermediate language.

$$\begin{aligned}
X_{\mathsf{completed}}[\![N]\!] &= N \\
X_{\mathsf{escaped}}[\![N]\!] &= N \\
X_{\mathsf{completed}}[\![A_1 \ \mathsf{and}\ A_2]\!] &= X_{\mathsf{completed}}[\![A_1]\!] +\!\!+ X_{\mathsf{completed}}[\![A_2]\!] \\
X_{\mathsf{escaped}}[\![A_1 \ \mathsf{and}\ A_2]\!] &= X_{\mathsf{escaped}}[\![A_1]\!] \cup X_{\mathsf{escaped}}[\![A_2]\!] \\
X_{\mathsf{completed}}[\![A_1 \ \mathsf{then}\ A_2]\!] &= X_{\mathsf{completed}}[\![A_1]\!] X_{\mathsf{completed}}[\![A_2]\!] \\
X_{\mathsf{escaped}}[\![A_1 \ \mathsf{then}\ A_2]\!] &= X_{\mathsf{escaped}}[\![A_1]\!] \cup (X_{\mathsf{completed}}[\![A_1]\!] X_{\mathsf{escaped}}[\![A_2]\!]) \\
X_{\mathsf{completed}}[\![A_1 \ \mathsf{or}\ A_2]\!] &= X_{\mathsf{completed}}[\![A_1]\!] \cup X_{\mathsf{completed}}[\![A_2]\!] \\
X_{\mathsf{escaped}}[\![A_1 \ \mathsf{or}\ A_2]\!] &= X_{\mathsf{escaped}}[\![A_1]\!] \cup X_{\mathsf{escaped}}[\![A_2]\!] \\
X_{\mathsf{completed}}[\![A_1 \ \mathsf{trap}\ A_2]\!] &= (X_{\mathsf{escaped}}[\![A_1]\!] X_{\mathsf{completed}}[\![A_2]\!]) \cup X_{\mathsf{completed}}[\![A_1]\!] \\
X_{\mathsf{escaped}}[\![A_1 \ \mathsf{trap}\ A_2]\!] &= X_{\mathsf{escaped}}[\![A_1]\!] X_{\mathsf{escaped}}[\![A_2]\!]
\end{aligned}$$

In the following we write set comprehensions for certain constant quad languages. To avoid too many trivial details we employ a few conventions. Entries in matrices in quads are assumed to be indep unless otherwise specified. Stating that a quad belongs to Q_{mn} constrains the size of quads in the set. We write () for empty vectors and empty matrices (where one of the dimensions is 0). The identity quad is named Id.

$$\begin{aligned}
X_{\mathsf{completed}}[\![\mathsf{complete}]\!] &= \{((),(),\mathbf{0},\mathbf{0}) \in Q_{n0} \mid n \geq 0\} \\
X_{\mathsf{escaped}}[\![\mathsf{complete}]\!] &= \emptyset \\
X_{\mathsf{completed}}[\![\mathsf{escape}]\!] &= \emptyset \\
X_{\mathsf{escaped}}[\![\mathsf{escape}]\!] &= \{\mathrm{Id} \in Q_{nn} \mid n \geq 0\}
\end{aligned}$$

$$
\begin{aligned}
X_{\mathsf{completed}}[\![\mathsf{fail}]\!] &= \emptyset \\
X_{\mathsf{escaped}}[\![\mathsf{fail}]\!] &= \emptyset \\
X_{\mathsf{completed}}[\![\mathsf{give}\ Y]\!] &= X_{\mathsf{Y}}[\![Y]\!] \\
X_{\mathsf{escaped}}[\![\mathsf{give}\ Y]\!] &= \emptyset \\
X_{\mathsf{completed}}[\![\mathsf{check}\ Y]\!] &= X_{\mathsf{Y}}[\![Y]\!]\{((),(),(\overline{\mathsf{bool}}),\mathbf{0}) \in Q_{10}\} \\
X_{\mathsf{escaped}}[\![\mathsf{check}\ Y]\!] &= \emptyset \\
X_{\mathsf{completed}}[\![\mathsf{enact\ application}\ Y_1\ \mathsf{to}\ Y_2]\!] &= X_{\mathsf{Y}}[\![Y_1]\!] @ X_{\mathsf{Y}}[\![Y_2]\!] \\
X_{\mathsf{escaped}}[\![\mathsf{enact\ application}\ Y_1\ \mathsf{to}\ Y_2]\!] &= X_{\mathsf{Y}}[\![Y_1]\!] @ X_{\mathsf{Y}}[\![Y_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
X_{\mathsf{Y}}[\![N]\!] &= N \\
X_{\mathsf{Y}}[\![\mathsf{it}]\!] &= \{(\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{0}) \in Q_{11}\} \\
X_{\mathsf{Y}}[\![\mathsf{them}]\!] &= \{\mathrm{Id} \in Q_{nn} \mid n \geq 0\} \\
X_{\mathsf{Y}}[\![\mathsf{sum}\ Y_1]\!] &= X_{\mathsf{Y}}[\![Y_1]\!]\{((\mathsf{int}),A,\mathbf{0},\mathbf{0}) \in Q_{n1} \mid A_{i1} = \overline{\mathsf{int}}, i \leq n, n \geq 0\} \\
X_{\mathsf{Y}}[\![\mathsf{the\ given}\ S]\!] &= \{(\mathbf{0},A,\mathbf{0},\mathbf{0}) \in Q_{nn} \mid n \in |\sigma[\![S]\!]|, \\
&\ A_{ii} = \overline{(\mathsf{typeof}(\sigma[\![S]\!])_n)_i}\} \\
X_{\mathsf{Y}}[\![\mathsf{the\ given}\ S\#k]\!] &= \{(\mathbf{0},A,\mathbf{0},\mathbf{0}) \in Q_{n1} \mid A_{k1} = \overline{\mathsf{typeof}(\sigma[\![S]\!])_1}, n \geq k\} \\
X_{\mathsf{Y}}[\![(Y_1,Y_2)]\!] &= X_{\mathsf{Y}}[\![Y_1]\!] +\!\!+ X_{\mathsf{Y}}[\![Y_2]\!] \\
X_{\mathsf{Y}}[\![b]\!] &= \{((\mathsf{bool}),\mathbf{0},\mathbf{0},\mathbf{0}) \in Q_{n1} \mid n \geq 0\} \\
X_{\mathsf{Y}}[\![n]\!] &= \{((\mathsf{int}),\mathbf{0},\mathbf{0},\mathbf{0}) \in Q_{n1} \mid n \geq 0\} \\
X_{\mathsf{Y}}[\![Y_1\ \mathsf{is}\ Y_2]\!] &= (X_{\mathsf{Y}}[\![Y_1]\!] +\!\!+ X_{\mathsf{Y}}[\![Y_2]\!])\{((\mathsf{bool}),\mathbf{0},\mathbf{1},\mathbf{0}) \in Q_{n1} \mid n \geq 0\} \\
X_{\mathsf{Y}}[\![\mathsf{abstraction\ of}\ N]\!] &= \lambda N
\end{aligned}
$$

6.8 Examples

Example 7 Consider the following fragment of an action semantics:

> eval i:integer = give integer-value of i .
>
> eval$[\![E_1$ "+" $E_2]\!]$ =
> eval E_1 and eval E_2 then
> | give sum (the given integer#1, the given integer#2) .

Via the $X_{\mathsf{completed}}$ translation this is translated to the following set of equations:

$$
\begin{aligned}
E &= \bigcup_n \{(\mathsf{int},\mathbf{0},\mathbf{0},\mathbf{0}) \in Q_{n1}\} \\
E &= (E +\!\!+ E) \cdot (G_1 +\!\!+ G_2) \cdot S
\end{aligned}
$$

where the first equation for E stems from the first definition for eval, and the second equation from the second definition. G_1 and G_2 correspond to the two parts of the argument tuple to sum, and S corresponds to the sum yielder. The sets G_1, G_2, and S are defined as follows (see also the definition of $X_{\mathsf{completed}}$):

$$G_k = \bigcup_{n \geq k} \{(\mathbf{0}, A, \mathbf{0}, \mathbf{0}) \in Q_{n1} \mid A_{k1} = \overline{\mathsf{int}}\}$$
$$S = \bigcup_{n \geq 0} \{(\mathsf{int}, A, \mathbf{0}, \mathbf{0}) \in Q_{n1} \mid A_{i1} = \overline{\mathsf{int}}, i \leq n\}$$

The desired solution to the equations is the least set of quads E satisfying the equations. We can compute this by iteration, starting from the empty set, yielding the solution:

$$E = \bigcup_{n} \{(\mathsf{int}, \mathbf{0}, \mathbf{0}, \mathbf{0}) \in Q_{n1}\}.$$

The information embedded in this set of quads is that the semantic function eval yields actions that always gives a single integer as output, and ignores the data given to it.

The above set of quads is of course infinite, and thus this direct approach doesn't work in practice. Instead the infinite sets are approximated by grids. ◇

The next example considers an action capable of giving tuples of more than one length.

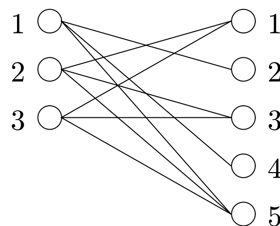
Example 8 Consider the action:

```
unfolding
| | | give the given integer#2
| | and
| | | give the given truth-value#1
| | and
| | | unfold
| or
| | give the given abstraction#3.
```

The recurrence generated by the simplification and the $X_{\mathsf{completed}}$ translation for this action has the form:

$$E = (E_1 + E_2 + E) \cup E_3.$$

This is solved by iteration as above, starting with $E = \emptyset$. Suppose 5-grids are chosen to represent the quad languages here. Let G be the grid representing the solution to the above recurrence. The figure below illustrates the quad G_{35} of the solution.



For typographical reasons, we do not write the **DAlg** elements on the figure, but all edges starting from input node 1 should be labeled with a bool may-dependence, and correspondingly, all edges starting from input nodes 2 and 3 should be labeled with int and abstraction may-dependences, respectively. ◇

6.9 The Flow Analysis

The flow analysis is a combined control flow analysis and type inference, as the two are interdependent. The goal of the analysis is to compute a set of quads corresponding to the data flow of each semantic function (non-terminal). As we have to deal with higher order functions, we use the control flow part of the analysis to approximate the sets of functions that may be called at every application point.

The control flow part is inspired by the work on control flow analysis and closure analysis for λ-calculi by Jones, Shivers and others [Jon81b, Jon81a, Shi88, Shi91]. The control flow analysis for action semantics is made difficult by the more complicated data flow of action notation. In λ-calculi it is easy to associate the use of a variable with its binding λ using simple scope rules or an initial α-renaming of the program. In action notation individual data items are not named, but flow as transient data, which needs to be traced through the action.

The simplification phase ensured that every syntactic abstraction is associated with a unique non-terminal. This is now used to identify each abstraction, just as each syntactic λ is labeled in traditional control flow analysis.

The analysis is formulated as a big step transition relation of the form:

$$(pt, qt, V, E) \triangleright (pt', qt', V', L),$$

where pt and pt' are *parameter tables* mapping non-terminals to sets of possible input vectors, such that $pt(N)$ is an approximation of the types of vectors that may be given to occurrences of the non-terminal N. The tables qt and qt' are *quad tables* mapping non-terminals to sets of quads. The quad language $qt(N)$ is an approximation of the data flow through the action term identified by the non-terminal N. V is an approximation of the types of vectors given as input to the expression E and V' is the corresponding approximation of the output vectors given as output by the expression. The language L is the set of quads modeling the data flow through the expression. Formally:

$$\begin{array}{rcl} pt, pt' & : & NT \to 2^{\text{Vec}} \\ qt, qt' & : & NT \to 2^{\text{Quad}} \\ V, V' & \subseteq & \text{Vec} \\ L & \subseteq & \text{Quad} \\ S & : & NT \to \mathit{Action} \cup \mathit{Yielder} \cup NT \end{array}$$

where Vec is the set of all type vectors, Quad is the set of all quadruples over our dependence algebra, and S is the set of simplified semantic equations. The maps pt and qt are extended to sets of non-terminals T as $qt(T) = \bigcup_{N \in T} qt(N)$. And we write $pt[T \mapsto V]$ for $pt[N_1 \mapsto V, N_2 \mapsto V, \ldots]$, where $T = \{N_1, N_2, \ldots\}$.

A set of vectors is mapped to a set of labels of syntactic abstractions by the map nt: $2^{\text{Vec}} \to 2^{NT}$, as:

$$\text{nt}(V) = \{N \in NT \mid \exists v \in V : |v| = 1, \mathsf{abs}_N \sqsubseteq v\}.$$

We extend maps with the following notation:

$$F[S_1 \mapsto S_2](x) = \mathsf{if}\ x \in S_1\ \mathsf{then}\ S_2\ \mathsf{else}\ F(x).$$

The following ordering on functions with sets as co-domains is used to express monotonicity of the analysis phase.

Definition 6.2 Let $f : A \to 2^B$ and $g : C \to 2^B$ be partial functions. Define the ordering:

$$f \subseteq g \iff \mathrm{dom}(f) \subseteq \mathrm{dom}(g) \text{ and } \forall x \in \mathrm{dom}(f) : f(x) \subseteq g(x).$$

□

Let L_T be the quad language associated with the terminal node T, as defined in the definition of the translation functions. The rules defining the transition relation are defined inductively on the structure of E. In reality there are two similar derivation systems, one for $t = \mathsf{completed}$, and one for $t = \mathsf{escaped}$.

$$\frac{}{(pt, qt, V, T) \triangleright (pt, qt, \mathrm{img}(V, L_T), L_T)}$$

$$\frac{(pt[N \mapsto pt(N) \cup V], qt, pt(N) \cup V, X_t[\![S(N)]\!]) \triangleright (pt', qt', V', L)}{(pt, qt, V, N) \triangleright (pt', qt'[N \mapsto qt'(N) \cup L], V', L \cup qt'(N))}$$

$$\frac{(pt, qt, V, E_1) \triangleright (pt', qt', V', L') \quad (pt', qt', V', E_2) \triangleright (pt'', qt'', V'', L'')}{(pt, qt, V, E_1 E_2) \triangleright (pt'', qt'', V'', L' L'')}$$

$$\frac{(pt, qt, V, E_1) \triangleright (pt', qt', V', L') \quad (pt', qt', V, E_2) \triangleright (pt'', qt'', V'', L'')}{(pt, qt, V, E_1 +\!\!+ E_2) \triangleright (pt'', qt'', V' +\!\!+ V'', L' +\!\!+ L'')}$$

$$\frac{(pt, qt, V, E_1) \triangleright (pt', qt', V', L') \quad (pt', qt', V, E_2) \triangleright (pt'', qt'', V'', L'')}{(pt, qt, V, E_1 \cup E_2) \triangleright (pt'', qt'', V' \cup V'', L' \cup L'')}$$

$$\frac{(pt, qt, pt(N), X_t[\![S(N)]\!]) \triangleright (pt', qt', V', L')}{(pt, qt, V, \lambda N) \triangleright (pt', qt'[N \mapsto qt'(N) \cup L'], \{\mathsf{abs}_N\}, \{(\mathsf{abs}_N, \mathbf{0}, \mathbf{0}, \mathbf{0})\})}$$

$$\frac{\begin{array}{rcl} (pt, qt, V, E_1) & \triangleright & (pt', qt', V', L') \\ (pt', qt', V, E_2) & \triangleright & (pt'', qt'', V'', L'') \quad T = \mathrm{nt}(V') \end{array}}{(pt, qt, V, E_1 @ E_2) \triangleright (pt''[T \mapsto V'' \cup pt''(T)], qt'', \mathrm{img}(V'', qt''(T)), L'' \cdot qt''(T))}$$

It is not hard to see that the parameter and quad tables grow monotonically during the analysis, as stated in the lemma below.

Lemma 6.3 *If* $(pt, qt, V, E) \triangleright (pt', qt', V', L)$ *then*

$$pt \subseteq pt' \text{ and } qt \subseteq qt'.$$

Inspecting the transition rules for the flow analysis it is not hard to see that the analysis can be implemented by repeated traversals of the expression tree E, and maintaining pt and qt as global variables, starting from the bottom of the grid ordering (p. 81). The only problem from an implementation point of view is that the sets of quads and vectors may grow to infinite size. This problem is handled by considering *grids* as defined in Chapter 4 instead of sets, and the formalities of this are treated in Section 6.11.

6.10 Soundness

The analysis computes a quad language corresponding to each semantic function. Soundness of the analysis means that if we apply the semantic function f to a concrete source language term t and obtain the action $A = f(t)$, and if A executes to completion or escapes then the data given to A must have satisfied the requirements embedded in the quad language L computed by the analysis, and the data given by A fulfills the guarantees about output that are also embedded in L.

In the following we abuse the notation a bit and write $\mathsf{typeof}(v)$ for $\mathsf{typeof}(v)_{|v|}$ whenever v is a single value. This abuse of notation is what makes the equation

$$\mathsf{typeof}(w_1 + w_2) = \mathsf{typeof}(w_1) + \mathsf{typeof}(w_2)$$

hold, as there is no concatenation defined on the result of $\mathsf{typeof}()$ in general. Also note that we do not make explicit the obvious injection from L to $\overline{L}_{\mathsf{indep}}$ taking a to its bare representation in $\overline{L}_{\mathsf{indep}}$.

Notice that the $\mathsf{unfolding}$ action carried around by the operational semantics is superfluous for simplified actions, and thus not used in the soundness proof. Also recall the dual use of $\sqsubseteq$ for subsort inclusion, and for ordering in the **DAlg**-structure. The symbol $\leq$ is used only for the type lattice ordering.

Theorem 6.4 (Soundness) *Let Y and A be a simplified yielder and action, respectively. Let $t \in \{\mathsf{completed}, \mathsf{escaped}\}$, and let S be the semantic equations. If there exists an $s \in V$ such that $\mathsf{typeof}(v) \sqsubseteq s$, and either*

$$(Y, B, v) \to w, \ \textit{and} \ (pt, qt, V, X_{\mathsf{Y}}[\![Y]\!]) \rhd (pt, qt, V', L)$$

or

$$(A, B, v) \to (t, w), \ \textit{and} \ (pt, qt, V, X_t[\![A]\!]) \rhd (pt, qt, V', L)$$

then there is a quad $Q \in L \cap Q_{|v||w|}$ such that

$$\begin{aligned} \mathsf{typeof}(w) &\sqsubseteq \mathrm{img}(\mathsf{typeof}(v), Q), &(6.7)\\ \mathsf{typeof}(v) &\leq \mathrm{rev}(Q, \mathsf{typeof}(w)), &(6.8)\\ \exists s' \in V' &: \mathsf{typeof}(w) \sqsubseteq s'. &(6.9)\end{aligned}$$

We express the above three properties concisely by writing

$$L, V' \models v, Q, w.$$

Before proving the soundness theorem we first establish a few lemmas that will allow us to reason at the level of sequents of the form $L, V' \models v, Q, w$.

Lemma 6.5 *The following rule is admissible.*

$$\frac{L_1, V_1 \models v, Q_1, w_1 \quad L_2, V_2 \models v, Q_2, w_2}{L_1 + L_2, V_1 + V_2 \models v, Q_1 + Q_2, w_1 + w_2}$$

Proof. Clearly $Q_1 +\!\!+ Q_2 \in L_1 +\!\!+ L_2$. For (6.7) we use Theorem 4.28 to compute:

$$\begin{array}{rcl} \mathrm{img}(\mathsf{typeof}(v), Q_1 +\!\!+ Q_2) & = & \mathrm{img}(\mathsf{typeof}(v), Q_1) +\!\!+ \mathrm{img}(\mathsf{typeof}(v), Q_2) \\ & \sqsupseteq & \mathsf{typeof}(w_1) +\!\!+ \mathsf{typeof}(w_2). \end{array}$$

For (6.8) we use Theorem 4.77 to compute:

$$\begin{array}{rcl} \mathsf{typeof}(v) & \leq & \mathrm{rev}(Q_1, \mathsf{typeof}(w_1)) \wedge \mathrm{rev}(Q_2, \mathsf{typeof}(w_2)) \\ & \leq & \mathrm{rev}(Q_1 +\!\!+ Q_2, \mathsf{typeof}(w_1 +\!\!+ w_2)). \end{array}$$

For (6.9) we use the existence of $s_1 \in V_1$, and $s_2 \in V_2$ such that: $\mathsf{typeof}(w_1) \sqsubseteq s_1$ and $\mathsf{typeof}(w_2) \sqsubseteq s_2$, to get:

$$\begin{array}{rcl} \mathsf{typeof}(w_1 +\!\!+ w_2) & = & \mathsf{typeof}(w_1) +\!\!+ \mathsf{typeof}(w_2) \\ & \sqsubseteq & s_1 +\!\!+ s_2 \in V_1 +\!\!+ V_2. \end{array}$$

□

Lemma 6.6 *The following rule is admissible.*

$$\frac{L_1, V_1 \models v, Q_1, w_1 \quad L_2, V_2 \models w_1, Q_2, w_2}{L_1 \cdot L_2, V_2 \models v, Q_1 Q_2, w_2}$$

Proof. By definition we have $Q_1 Q_2 \in L_1 \cdot L_2$. For (6.7) we use Theorem 4.16 to compute:

$$\begin{array}{rcl} \mathrm{img}(\mathsf{typeof}(v), Q_1 Q_2) & = & \mathrm{img}(\mathrm{img}(\mathsf{typeof}(v), Q_1), Q_2) \\ & \sqsupseteq & \mathrm{img}(\mathsf{typeof}(w_1), Q_2) \\ & \sqsupseteq & \mathsf{typeof}(w_2). \end{array}$$

For (6.8) we use Theorem 4.73 to compute:

$$\begin{array}{rcl} \mathrm{rev}(Q_1 Q_2, \mathsf{typeof}(w_2)) & \geq & \mathrm{rev}(Q_1, \mathrm{rev}(Q_2, \mathsf{typeof}(w_2))) \\ & \geq & \mathrm{rev}(Q_1, \mathsf{typeof}(w_1)) \\ & \geq & \mathsf{typeof}(v). \end{array}$$

Property (6.9) follows from the second premiss: there is a $s_2 \in V_2$ such that $\mathsf{typeof}(w_2) \sqsubseteq s_2$. □

Lemma 6.7 *The following rule is admissible.*

$$\frac{L_1, V_1 \models v, Q_1, w_1 \quad L_1 \subseteq L_2 \quad V_1 \subseteq V_2}{L_2, V_2 \models v, Q_1, w_1}$$

Proof. Trivial from the definitions. □

Proof. (Soundness theorem.) Notice that the parameter table (pt) and quad table (qt) are the same on both sides of the $\triangleright$ relation, so that the analysis has reached a fixed point with respect to these tables. Using Lemma 6.3, if the tables are the same on both sides in the consequent of a rule, they are necessarily the same on both sides of the $\triangleright$ symbol in the antecedents of the rule.

The proof goes by induction in the derivation of $(Y, B, v) \to w$, respectively $(A, B, v) \to (t, w)$. During the proof we also ensure that the two following global properties hold by induction in the derivation of the $\triangleright$ relation:

1. All @-nodes in the derivation tree of the $\triangleright$ relation have the form:
$$\frac{(pt, qt, V_0, E_1) \triangleright (pt, qt, V_0', L_0) \quad (pt, qt, V_0, E_2) \triangleright (pt, qt, V_0'', L_0')}{(pt, qt, V_0, E_1@E_2) \triangleright (pt, qt, \mathrm{img}(V_0'', qt(\mathrm{nt}(V_0'))), L_0' \cdot qt(\mathrm{nt}(V_0')))}$$
and $\forall N \in \mathrm{nt}(V_0') : V_0'' \subseteq pt(N)$.

2. All λN-nodes in the derivation tree of the $\triangleright$ relation have the form:
$$\frac{(pt, qt, pt(N), X_t[\![S(N)]\!]) \triangleright (pt, qt, V_1', L_1')}{(pt, qt, V_1, \lambda N) \triangleright (pt, qt, \{\mathsf{abs}_N\}, \{(\mathsf{abs}_N, \mathbf{0}, \mathbf{0}, \mathbf{0})\})}$$
and $L_1' \subseteq qt(N)$.

Both of these properties are easy consequences of the rules for the $\triangleright$ relation. The proof now proceeds by a case analysis on Y and A. We show only the most interesting and illustrative cases.

- $Y = \mathsf{the\ given}\ S\#k$: Here we have $(Y, B, v) \to v_k$, where $v_k \sqsubseteq \sigma[\![S]\!]$. Let $Q = (\mathbf{0}, A, \mathbf{0}, \mathbf{0}) \in Q_{n1}$, where $n = |v|$, and $A_{k1} = \overline{\mathsf{typeof}(\sigma[\![S]\!])_1}$.

  Using the definition of rev() and Lemma 4.69 we first compute for (6.8):
$$\begin{aligned}
\mathrm{rev}(Q, \mathsf{typeof}(v_k))_i &= \mathrm{rev}(A, \mathsf{typeof}(v_k))_i \wedge \Psi(\mathbf{0}) \\
&= \Psi(A_{i1}\overline{\mathsf{typeof}(v_k)_1}) \\
&= \begin{cases} \top & \text{if } i \neq k \\ \mathsf{typeof}(\sigma[\![S]\!])_1 \wedge \mathsf{typeof}(v_k) & \text{if } i = k \end{cases} \\
&\geq \mathsf{typeof}(v).
\end{aligned}$$
  For the (6.7) we compute:
$$\begin{aligned}
\mathrm{img}(\mathsf{typeof}(v), Q) &= \mathsf{typeof}(v)A \\
&= \mathsf{typeof}(v_k) \wedge \mathsf{typeof}(\sigma[\![S]\!])_1 \\
&= \mathsf{typeof}(v_k \sqcap \sigma[\![S]\!])_1 \\
&\sqsupseteq \mathsf{typeof}(v_k).
\end{aligned}$$
  By definition $V' = \mathrm{img}(V, L_T)$, and by assumption there is an $s \in V$ such that $\mathsf{typeof}(v) \sqsubseteq s$, so $s_k \in V'$, and clearly $\mathsf{typeof}(v_k) \sqsubseteq s_k$. This ensures (6.9).

- $Y = Y_1 \text{ is } Y_2$: From the operational semantics we have $(Y, B, v) \to w$ only if $(Y_1, B, v) \to w_1$, $(Y_2, B, v) \to w_2$, and $\mathsf{typeof}(w) = \mathsf{bool}$.

  The translation yields:

$$X_{\mathsf{Y}}[\![Y_1 \text{ is } Y_2]\!] = (X_{\mathsf{Y}}[\![Y_1]\!] + X_{\mathsf{Y}}[\![Y_2]\!])\{((\mathsf{bool}), \mathbf{0}, \mathbf{1}, \mathbf{0}) \in Q_{n1} \mid n \geq 0\}$$

  By the induction hypothesis and the definition of the $\triangleright$ relation we must have:

$$\begin{aligned} (pt, qt, V, X_{\mathsf{Y}}[\![Y_1]\!]) &\triangleright (pt, qt, V_1, L_1), \\ (pt, qt, V, X_{\mathsf{Y}}[\![Y_2]\!]) &\triangleright (pt, qt, V_2, L_2), \\ (pt, qt, V_1 + V_2, \{((\mathsf{bool}), \mathbf{0}, \mathbf{1}, \mathbf{0}) \in Q_{n1} \mid n \geq 0\}) &\triangleright (pt, qt, V', L), \end{aligned}$$

  and that there are $Q_1 \in L_1$ and $Q_2 \in L_2$ such that:

$$L_1, V_1 \models v, Q_1, w_1 \quad \text{and} \quad L_2, V_2 \models v, Q_2, w_2.$$

  By the definition of $\triangleright$ we have:

$$L = (L_1 + L_2)\{((\mathsf{bool}), \mathbf{0}, \mathbf{1}, \mathbf{0}) \in Q_{n1} \mid n \geq 0\},$$

  and

$$V' = \mathrm{img}(V_1 + V_2, \{((\mathsf{bool}), \mathbf{0}, \mathbf{1}, \mathbf{0}) \in Q_{n1} \mid n \geq 0\}). \tag{6.10}$$

  Let $Q = ((\mathsf{bool}), \mathbf{0}, \mathbf{1}, \mathbf{0}) \in Q_{n1}$, where $n = |w_1| + |w_2|$. Using Theorems 4.73 and 4.77, and the induction hypothesis we compute:

$$\begin{aligned} \mathrm{rev}((Q_1 + Q_2)Q, \mathsf{bool}) &\geq \mathrm{rev}(Q_1 + Q_2, \mathrm{rev}(Q, \mathsf{bool})) \\ &= \mathrm{rev}(Q_1 + Q_2, \mathrm{rev}(\mathbf{0}, \mathsf{bool}) \wedge \Psi(\mathbf{1})) \\ &= \mathrm{rev}(Q_1 + Q_2, \prod_{i \leq n} \top) \\ &\geq \mathrm{rev}(Q_1, \prod_{i \leq |w_1|} \top) \wedge \mathrm{rev}(Q_2, \prod_{j \leq |w_2|} \top) \\ &\geq \mathrm{rev}(Q_1, \mathsf{typeof}(w_1)) \wedge \mathrm{rev}(Q_2, \mathsf{typeof}(w_2)) \\ &\geq \mathsf{typeof}(v). \end{aligned}$$

  For (6.7) we use Theorem 4.16:

$$\begin{aligned} \mathrm{img}(\mathsf{typeof}(v), (Q_1 + Q_2)Q) &= \mathrm{img}(\mathrm{img}(\mathsf{typeof}(v), Q_1 + Q_2), Q) \\ &= \mathrm{img}(\mathsf{typeof}(v), Q_1 + Q_2)\mathbf{0} + \mathsf{bool} \\ &= \mathsf{bool} \qquad\qquad\qquad\qquad (6.11) \\ &\sqsupseteq \mathsf{typeof}(w). \end{aligned}$$

  Together, equations (6.10) and (6.11) ensure property (6.9).

- $Y = (Y_1, Y_2)$: From the operational semantics we have $((Y_1, Y_2), B, v) \to w_1 + w_2$ whenever $(Y_1, B, v) \to w_1$, and $(Y_2, B, v) \to w_2$. As

$$X_{\mathsf{Y}}[\![(Y_1, Y_2)]\!] = X_{\mathsf{Y}}[\![Y_1]\!] + X_{\mathsf{Y}}[\![Y_2]\!]$$

we have (from the definition of the rules for the $\triangleright$ relation):

$$\begin{array}{lcl}(pt, qt, V, X_{\mathsf{Y}}[\![Y_1]\!]) & \triangleright & (pt, qt, V_1, L_1), \\ (pt, qt, V, X_{\mathsf{Y}}[\![Y_2]\!]) & \triangleright & (pt, qt, V_2, L_2),\end{array}$$

$L = L_1 + L_2$, and $V' = V_1 + V_2$.

By the induction hypothesis we get $L_1, V_1 \models v, Q_1, w_1$ and $L_2, V_2 \models v, Q_2, w_2$. Using Lemma 6.5 we get $L_1 + L_2, V_1 + V_2 \models v, Q_1 + Q_2, w_1 + w_2$ which is the desired result.

- $Y = b$: From the operational semantics we have that $(b, B, v) \to w$ where $\mathsf{typeof}(w) = \mathsf{bool}$ regardless of v. Let $Q = ((\mathsf{bool}), \mathbf{0}, \mathbf{0}, \mathbf{0}) \in Q_{n1}$ for $n = |v|$. We have

$$\mathrm{rev}(Q, \mathsf{bool}) = \mathrm{rev}(\mathbf{0}, v) \wedge \Psi(\mathbf{0}) = \prod_{i \leq n} \top.$$

and

$$\mathrm{img}(v, Q) = v\mathbf{0} + (\mathsf{bool}) = (\mathsf{bool}).$$

Property (6.9) holds by the definition of $\triangleright$ for terminals.

- $Y = \mathsf{abstraction\ of}\ N$: By the operational semantics we have $(Y, B, v) \to \langle N, B \rangle$, and we have $V' = \{\mathsf{abs}_N\}$, so we have $V' \ni s' = \mathsf{abs}_N = \mathsf{typeof}(\langle N, B \rangle)$ which shows (6.9). Also $\mathsf{abs}_N = \mathrm{img}(\mathsf{typeof}(v), (\mathsf{abs}_N, \mathbf{0}, \mathbf{0}, \mathbf{0}))$ proving (6.7). For (6.8) we have $\top = \mathrm{rev}((\mathsf{abs}_N, \mathbf{0}, \mathbf{0}, \mathbf{0}), \mathsf{typeof}(w))$. The definition of $\triangleright$ ensures that the first global property holds.

- $Y = N$: By the extended semantics we have $(N, B, v) \to w$ because $(S(N), B, v) \to w$, and we have $(pt, qt, V, N) \triangleright (pt, qt, V', L'' \cup qt(N))$ because $(pt, qt, pt(N) \cup V, X_{\mathsf{Y}}[\![S(N)]\!]) \triangleright (pt, qt, V', L'')$. By the induction hypothesis we have $L'', V' \models v, Q, w$. As $L'' \subseteq L$ the desired result holds by Lemma 6.7.

For completing actions, $t = \mathsf{completed}$:

- $A = \mathsf{complete}$: Recall the definition of $X_{\mathsf{completed}}[\![\mathsf{complete}]\!]$. By the convention that the intersection of an empty family of sets is the largest set, we have $\mathrm{rev}(((), (), \mathbf{0}, \mathbf{0}), w) = \top$, so property (6.8) is fulfilled. For (6.7) it is clear that the image of v through the quad yields an empty tuple which ensures the inequality.

- $A = \mathsf{fail}$: This primitive never completes so the case holds vacuously.

- $A = \mathsf{escape}$: As the above case.

- $A = A_1\ \mathsf{and}\ A_2$: This follows as the case for yielders where $Y = (Y_1, Y_2)$.

- $A = A_1\ \mathsf{or}\ A_2$: Immediate by induction.

- $A = A_1$ $\textsf{then}$ A_2: By induction and Lemma 6.6. See also the case for $\textsf{trap}$ below.

- $A = A_1$ $\textsf{trap}$ A_2: If $(A_1\ \textsf{trap}\ A_2, B', v) \to (\textsf{completed}, w)$ then we must have that either

$$(A_1, B', v) \to (\textsf{completed}, w)$$

or

$$(A_1, B', v) \to (\textsf{escaped}, w') \text{ and } (A_2, B', w') \to (\textsf{completed}, w).$$

The translation yields:

$$X_{\textsf{completed}}[\![A_1\ \textsf{trap}\ A_2]\!] = X_{\textsf{escaped}}[\![A_1]\!]X_{\textsf{completed}}[\![A_2]\!] \cup X_{\textsf{completed}}[\![A_1]\!],$$

thus

$$(pt, qt, V, X_{\textsf{completed}}[\![A_1\ \textsf{trap}\ A_2]\!]) \triangleright (pt, qt, V', L)$$

means that we must have all of the following by the definition of the analysis rules:

$$\begin{array}{rcl} (pt, qt, V, X_{\textsf{escaped}}[\![A_1]\!]X_{\textsf{completed}}[\![A_2]\!]) & \triangleright & (pt, qt, V_1, L_1) \\ (pt, qt, V, X_{\textsf{completed}}[\![A_1]\!]) & \triangleright & (pt, qt, V_2, L_2) \\ (pt, qt, V, X_{\textsf{escaped}}[\![A_1]\!]) & \triangleright & (pt, qt, V_3, L_3) \\ (pt, qt, V_3, X_{\textsf{completed}}[\![A_2]\!]) & \triangleright & (pt, qt, V_4, L_4), \end{array}$$

and $V' = V_4 \cup V_2$, and $L = L_2 \cup (L_3 \cdot L_4)$. There are now two cases:

1. The first action completes: By the induction hypothesis we get $L_2, V_2 \models v, Q_2, w$. As $L_2 \subseteq L$ and $V_2 \subseteq V'$ the desired results follows by Lemma 6.7.
2. The first action escapes: Here we get that $L_3, V_3 \models v, Q_3, w'$. As there exists an $s_3 \in V_3$ such that $\textsf{typeof}(w') \sqsubseteq s_3$ we can apply the induction hypothesis once more to get that $L_4, V_4 \models w', Q_4, w$. Using Lemma 6.6 we then get $L_3 \cdot L_4, V_4 \models v, Q_3Q_4, w$. And since $L_3 \cdot L_4 \subseteq L$ and $V_4 \subseteq V'$ we get the desired result via Lemma 6.7.

- $A = \textsf{check}\ Y$: By the evaluation rules we must have $(Y, B, v) \to w'$, with $\textsf{typeof}(w') = \textsf{bool}$. By the induction hypothesis we must have that $(pt, qt, V, X_{\textsf{Y}}[\![Y]\!]) \triangleright (pt, qt, V_1, L_1)$ and that there is a $Q \in L_1$ such that $L_1, V_1 \models v, Q, w'$, where $Q \in Q_{n1}$, with $n = |v|$. For the image we have:

$$\text{img}(\textsf{typeof}(v), Q((), (), \overline{\textsf{bool}}, \mathbf{0})) = ().$$

For the reverse image we have:

$$\begin{array}{rcl} \text{rev}(Q((), (), \overline{\textsf{bool}}, \mathbf{0}), ()) & \geq & \text{rev}(Q, \text{rev}(((), (), \overline{\textsf{bool}}, \mathbf{0}), ())) \\ & = & \text{rev}(Q, \text{rev}((), ()) \wedge \Psi(\overline{\textsf{bool}})) \\ & = & \text{rev}(Q, \textsf{bool}) \\ & \geq & \textsf{typeof}(v). \end{array}$$

Property (6.9) follows as V' contains the empty tuple.

- $A = \mathsf{enact\ application}\ Y_1\ \mathsf{to}\ Y_2$: The evaluation rule for this case looks as:

$$\frac{(Y_1, B, v) \to \langle N, B' \rangle \quad (Y_2, B, v) \to w_2 \quad (N, B', w_2) \to (t, w)}{(\mathsf{enact\ application}\ Y_1\ \mathsf{to}\ Y_2, B, v) \to (t, w)}$$

and for the analysis we have:

$$\frac{\begin{array}{c}(pt, qt, V, X_{\mathsf{Y}}[\![Y_1]\!]) \rhd (pt, qt, V_1, L_1) \\ (pt, qt, V, X_{\mathsf{Y}}[\![Y_2]\!]) \rhd (pt, qt, V_2, L_2) \quad T = \mathrm{nt}(V_1)\end{array}}{(pt, qt, V, X_{\mathsf{Y}}[\![Y_1]\!]@X_{\mathsf{Y}}[\![Y_2]\!]) \rhd (pt, qt, \mathrm{img}(V_2, qt(T)), L_2 \cdot qt(T))}$$

The first global property is ensured as $V_2 \subseteq pt(N)$ for all $N \in \mathrm{nt}(V_1)$.

By the induction hypothesis we have $L_1, V_1 \models v, Q_1, w_1$, where $w_1 = \langle N, B' \rangle$. In particular we have $\mathsf{abs}_N = \mathsf{typeof}(\langle N, B' \rangle) \sqsubseteq \mathrm{img}(\mathsf{typeof}(v), Q_1)$, and that there is a $s_1 \in V_1$ such that $\mathsf{abs}_N = \mathsf{typeof}(w_1) \sqsubseteq s_1$. By the definition of the function nt this means that $N \in \mathrm{nt}(V_1)$, and therefore $V_2 \subseteq pt(N)$.

By a second application of the induction hypothesis we get $L_2, V_2 \models v, Q_2, w_2$, in particular there is a $s_2 \in V_2$ such that $\mathsf{typeof}(w_2) \sqsubseteq s_2$.

By the second global property we have that $L_1' \subseteq qt(N)$ in

$$\frac{(pt, qt, pt(N), X_t[\![S(N)]\!]) \rhd (pt, qt, V_1', L_1')}{(pt, qt, V_1, \lambda N) \rhd (pt, qt, \{\mathsf{abs}_N\}, \{(\mathsf{abs}_N, \mathbf{0}, \mathbf{0}, \mathbf{0})\})}$$

By a third application of the induction hypothesis where we use:

$$(N, B', w_2) \to (t, w), \quad (pt, qt, pt(N), X_t[\![S(N)]\!]) \rhd (pt, qt, V_1', L_1'),$$

and the existence of $s_2 \in pt(N)$ such that $\mathsf{typeof}(w_2) \sqsubseteq s_2$, we get $L_1', V_1' \models w_2, Q', w$. To finish this case we then compute: for (6.7):

$$\begin{array}{rcl}\mathsf{typeof}(w) & \sqsubseteq & \mathrm{img}(\mathsf{typeof}(w_2), Q') \\ & \sqsubseteq & \mathrm{img}(\mathrm{img}(\mathsf{typeof}(v), Q_2), Q') \\ & = & \mathrm{img}(\mathsf{typeof}(v), Q_2 Q'),\end{array}$$

and for (6.8):

$$\begin{array}{rcl}\mathsf{typeof}(v) & \leq & \mathrm{rev}(Q_2, \mathsf{typeof}(w_2)) \\ & \leq & \mathrm{rev}(Q_2, \mathrm{rev}(Q', \mathsf{typeof}(w))) \\ & \leq & \mathrm{rev}(Q_2 Q', \mathsf{typeof}(w)).\end{array}$$

For (6.9) we use that $\exists s_2 \in V_2 : \mathsf{typeof}(w_2) \sqsubseteq s_2$, and that $Q' \in qt(\mathrm{nt}(V_1))$ which ensures that

$$\mathsf{typeof}(w) \sqsubseteq \mathrm{img}(\mathsf{typeof}(w_2), Q') \sqsubseteq \mathrm{img}(s_2, Q') \in \mathrm{img}(V_2, qt(\mathrm{nt}(V_1))).$$

For escaping actions, $t = \mathsf{escaped}$:

- $A = \mathsf{complete}$: This action never escapes.
- $A = \mathsf{escape}$: This case is trivial from the definitions.
- $A = A_1$ and A_2: This combined action escapes when A_1 or A_2 escapes. The case then follows by induction.
- $A = A_1$ then A_2: Similar to the completing case for trap.
- $A = A_1$ trap A_2: As for the completing case for then.

□

6.11 Finiteness

The above formulation of the type inference and the soundness theorem deals with quad languages that are in general infinite. In order to obtain an implementable algorithm we have to approximate these large sets with finite approximative representations. This is accomplished by using the abstraction of quad languages into grids developed in Section 4.8.

Instead of interpreting the flow analysis algorithm on page 151 as generating a language of quads we interpret it as generating a grid. Set union becomes l.u.b. on grids, and concatenation and composition of languages becomes concatenation and composition of grids. Correspondingly we use the $\sqsubseteq$ ordering on grids instead of subset inclusion.

The method of first formulating a program analysis using possibly infinite sets, followed by an abstraction to finite conservative representations of those sets is closely related to the method of abstract interpretation. In that method the analysis is first formulated by the so-called accumulating semantics which is then related to the abstract interpretation via an adjunction. Here we employ the adjunction between sets of quads and their representing grids.

As the abstraction from quad languages to grids is conservative (as proved in Section 4.8) we get that the grid generated by a non-terminal in the re-interpreted transition system represents a superset of the quad language generated by the original transition system. Recall the concretization function γ_K from Section 4.8 and extend it pointwise to maps. If we write $\triangleright'$ for the re-interpreted transition relation this can be written as follows: Suppose

$$(pt_1, qt_1, V_1, E) \triangleright (pt_2, qt_2, V_2, L) \text{ and } (pt'_1, qt'_1, V'_1, E) \triangleright' (pt'_2, qt'_2, V'_2, G)$$

where $pt'_1, pt'_2, qt'_1, qt'_2$ map to grids instead of sets of quads, and V'_1, V'_2 are grids as well. If

$$pt_1 \subseteq \gamma_K(pt'_1), qt_1 \subseteq \gamma_K(qt'_1), V_1 \subseteq \gamma_K(V'_1)$$

then

$$pt_2 \subseteq \gamma_K(pt'_2), qt_2 \subseteq \gamma_K(qt'_2), V_2 \subseteq \gamma_K(V'_2),$$

and $L \subseteq \gamma_K(G)$.

Theorem 6.8 (Soundness) *Let Y be a simplified yielder, and A be a simplified action. Let $t \in \{\mathsf{completed}, \mathsf{escaped}\}$, and let S be the semantic equations. If there exists an $s \in \gamma_K(V)$ such that* $\mathsf{typeof}(v) \sqsubseteq s$*, and either*

$$(Y, B, v) \to w, \ \ \textit{and}\ (pt, qt, V, X_{\mathsf{Y}}[\![Y]\!]) \rhd' (pt, qt, V', G)$$

or

$$(A, B, v) \to (t, w), \ \ \textit{and}\ (pt, qt, V, X_t[\![A]\!]) \rhd' (pt, qt, V', G)$$

then there is a $Q \in \gamma_K(G)$ such that

$$\begin{aligned} \mathsf{typeof}(w) &\sqsubseteq \mathrm{img}(\mathsf{typeof}(v), Q), && (6.12)\\ \mathsf{typeof}(v) &\leq \mathrm{rev}(Q, \mathsf{typeof}(w)), && (6.13)\\ \exists s' \in V' &: \mathsf{typeof}(w) \sqsubseteq s'. && (6.14) \end{aligned}$$

Proof. By the previous soundness theorem and Theorems 4.50, 4.51, and 4.52. □

As can be seen from the soundness theorems, a safe approximation of the input and output types of an action can be found using the image and reverse image operations. If G is the grid computed by the analysis for an action then the type of data given by the action can be approximated as: $\bigvee_{Q \in \gamma_K(G)} \mathrm{img}(\top, Q)$, and similarly for the input type: $\bigvee_{Q \in \gamma_K(G)} \mathrm{rev}(Q, \top)$.

Let n be the number of non-terminals in the simplified semantics. Inspecting each rule of the analysis on page 151, with the re-interpretation of languages as grids in mind, it is easy to see that the worst case cost of the application of a single rule is $O(nK^4 + K^6)$ in the application (@) case. Computing $qt''(T)$ where T is a set of size at most n may be done in n l.u.b. operations each costing $O(K^4)$.

Let h be the height of the type lattice. The height of the grid lattice is bounded by hK^4, as each quad entry in the grid has a height of at most hK^2. There is a grid for each non-terminal, so the height of the quad table is bounded by nhK^4. Correspondingly, the height of the parameter table (of vector-grids) is bounded by nhK^2. In order to reach the least fixed point with respect to the parameter- and quad-tables, the entire translated semantics needs to be traversed at most $nh(K^4 + K^2)$ times. Each traversal either finds a fixed point with respect to (pt, qt) or finds a pair (pt', qt') at least one step higher in the lattice by Lemma 6.3.

Let m be the size of the translated semantics (number of syntactic nodes). From the above considerations we get that the complexity of the analysis is in $O(m(nK^4 + K^6)(nh(K^4 + K^2)))$, that is $O(mnhK^{10} + (mn^2 + mnh)K^8 + mn^2K^6)$. As $n \leq m$ we can simplify this to $O(m^2hK^{10} + m^3K^8)$. In practice h and K are (small) constants, so overall the algorithm is cubic in the size of the semantics.

The grid size K was included in the above computations to show how much the grid size influences the cost of the analysis. Larger grids means more precise analysis, but at a non-trivial cost. In practice, most action semantics descriptions use only short tuples of length two or three, so grid sizes of three or four would typically suffice.

6.12 Summary

This chapter has presented a soft type inference system for action semantic equations based on the dependence quad technology developed in Chapter 4. The analysis computes a grid of quads corresponding to each semantic function from which information about data flow

and types can be gleaned. The analysis has been proved sound with respect to a natural operational semantics for action notation.

Chapter 7

Conclusion

7.1 Contributions

This thesis has introduced two kinds of program analysis: trust analysis and dependence algebra-based analysis. Chapter 2 introduced the concept of trust analysis, and gave two analyses for a structured imperative programming language with pointers. One based on abstract interpretation, and one based on constraint generation and solving. The two analyses were proved sound and compared to each other.

Chapter 3 gave the first trust analysis for an extension of the simply-typed λ-calculus in terms of an annotated type system. The extended calculus was proved Church-Rosser, and a subject-reduction theorem showing the soundness of the type system with respect to the calculus was shown. A cubic time type inference algorithm was given and shown sound and complete with respect to the type system.

Chapter 4 introduced the theory of dependence algebra. Matrices and quadruples were built on top of the basic dependence algebra, and various properties of the algebraic operations were proved. The important notion of the image of a vector through a quad was defined as well. The notion of quad languages was introduced, and an algebra of operations on quad languages was developed. Grids were introduced as a finite representation of infinite quad languages, and several theorems showing the soundness of the finite abstraction were given. A particular non-trivial construction of a dependence algebra from a distributive lattice was described, leading to the notion of a reverse image crucial for the computation of must/may dependences. Finally the theory was related to other analysis technologies.

Chapters 5 and 6 exhibited two examples of applications of dependence algebra: trust analysis of C and type inference for actions semantics. Chapter 5 described an implementation of a trust analysis for the C programming language based on the dependence technology developed in Chapter 4. Special care was taken to be able to handle unstructured control flow in the program point sensitive analysis. The back end of the analysis was proved sound with respect to an operational semantics of a stack machine.

Chapter 6 developed a cubic time soft type inference algorithm for action semantic equations, also based on the dependence algebra theory. Here all the machinery (grids, must/may dependence, reverse images) of Chapter 4 came into play. The analysis was proved sound with respect to a natural operational semantics for the functional (including higher-order abstractions) facet of action notation.

7.2 Suggestions for Future Research

There are many avenues open for further research in the areas of trust and dependence analysis. There is of course always a need for more precise, fast, and flexible analyses. There is, however, a more urgent need to gain experience in the practical use of trust analysis. As was mentioned in the discussion in Chapter 5, programs written without trust analysis in mind are often hard to change so that a trust analyzer may verify the safety of the program. This problem may be attacked on two fronts: better, more flexible, analyses would be able to verify more programs, and new programs written with the analysis in mind may be better suited for analysis. Writing new programs with trust in mind may well result in better programs, just as writing in a strongly typed language often results in better programs, due to the stronger discipline enforced by the type system.

The particular implementation of the trust analysis for C would benefit immensely from a good "points-to" analysis, but this is hard to combine with a modular analysis, analyzing each source file separately.

The theory of dependence algebra should be applied to more problems, to tests its powers, and the theory in itself could, as a result of the applications, be extended. One obvious candidate application for dependence algebra is the problem of strictness analysis for higher-order lazy functional languages.

The grid abstraction of sets of quads is pretty crude and better representations of such sets should be sought.

The type inference for action semantics equations should be extended to a more useful subset of action notation incorporating some part of data notation as well. The extended analysis should also be implemented to test its strength on real examples.

Bibliography

[Abr65] S. Abraham. Some questions of phrase-structure grammars. *Computational Linguistics*, 4:61–70, 1965.

[Agr94] Hiralal Agrawal. On slicing programs with jump statements. *ACM SIGPLAN Notices*, 29(6):302–312, June 1994. Proceedings of PLDI'94: Conf. on Programming Language Design and Implementation.

[AKG95] Hassan Aït-Kaci and Jacques Garrigue. Label-selective lambda-calculus: Syntax and confluence. *Theoretical Computer Science*, 151:353–383, 1995. URL: http://wwwfun.kurims.kyoto-u.ac.jp/%7Egarrigue/papers/.

[All70] Frances E. Allen. Control flow analysis. In *Proc. of a Symp. on Compiler Optimization*, volume 5 of *ACM SIGPLAN Notices*, pages 1–19, UIUC, July 1970. ACM, ACM Press.

[AM91] Alexander Aiken and Brian R. Murphy. Static Type Inference in a Dynamically Typed Language. In *POPL'91: Proc. of the 18'th annual ACM Symp. on Principles of Programming Languages*, pages 279–290, Orlando, FL, January 1991. ACM.

[AML+93] F. Anklesaria, M. McCahill, P. Lindner, D.J. Johnson, D. Torrey, and B. Alberti. *RFC-1436: The Internet Gopher Protocol (a distributed document search and retrieval protocol).* IETF, March 1993.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, Computer Science Department at University of Copenhagen, May 1994.

[AR79] Gregory R. Andrews and Richard P. Reitman. Certifying information flow properties of programs: An axiomatic approach. In *POPL'79: Proc. of the 6'th ACM Symp. on Principles of Programming Languages*, pages 283–290, San Antonio, TX, January 1979. ACM.

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. on Programming Languages and Systems*, 2(1):56–76, January 1980.

[ASD86] Alfred V. Aho, Ravi Sethi, and Jeffrey D.Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA'93: Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[Bar81] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1981.

[BBM94] Jean-Pierre Banâtre, Ciarán Bryce, and Daniel Le Metayer. Compile-time detection of information flow in sequential programs. In Dieter Gollmann, editor, *Computer Security – ESORICS 94, 3rd European Symp. on Research in Comp. Security*, volume 875 of *Lecture Notes in Computer Science*, pages 55–73, Brighton, UK, November 1994. Springer-Verlag.

[BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. on Programming Languages and Systems*, 7(1):37–61, January 1985.

[BH93] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control flow. In P. A. Fritzson, editor, *AADEBUG'93: Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 206–222, Linkö bing, May 1993. Springer-Verlag.

[Bir48] Garrett Birkhoff. *Lattice Theory*. American Mathematical Society, 1948. AMS Colloquium publications, volume XXV.

[BJP91] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

[BL73] D. E. Bell and L. J. LaPadula. Secure computer system: Mathematical foundations and model. Technical Report M74-244, MITRE Corp., 1973.

[BL76] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and MULTICS interpretation. Technical Report MTR-2997, rev. 1, MITRE Corp., Bedford, Mass, March 1976.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. *RFC-1945: Hypertext Transfer Protocol – HTTP/1.0*. IETF, May 1996.

[BMW92] Deryck F. Brown, Hermano Moura, and David A. Watt. Actress: an Action Semantics Directed Compiler Generator. In *Proceedings of the 1992 Workshop on Compiler Construction, Paderborn, Germany*, volume 641 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer-Verlag, 1984.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77: Conf. Proc. of the 4'th ACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[CC79a] Patrick Cousot and Radhia Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979.

[CC79b] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL'79: Proc. of the 6'th ACM Symp. on Principles of Programming Languages*, pages 269–282, San Antonio, TX, January 1979. ACM.

[CDG+89] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report TR-52, DEC-SRC, November 1989.

[CER94] CERT Advisory 94:12 Sendmail Vulnerability. Technical report, CERT, 1994. URL: ftp://ftp.cert.org/.

[CF94] J.-D. Choi and Jeanne Ferrante. Static slicing in the presence of GOTO statements. *ACM Letters on Programming Languages and Systems*, 1994.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *POPL'89: Proc. ACM Symp. on Principles of Programming Languages*, pages 25–. ACM, 1989.

[CFR91] Ron Cytron, Jeanne Ferrante, and Barry K. Rosen. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[DD77] Dorothy E. Denning and Peter J. Denning. Certifications of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–512, July 1977.

[Den76] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–242, May 1976.

[Den82] Dorothy Elisabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[Deu92] Alain Deutsch. *Operational Models of Programming Languages and Representation of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data*. PhD thesis, University of Paris, VI, LIX, Ecole Polytechnique, 1992.

[DP89] Jürgen Dassow and Gheorghe Păun. *Regulated Rewriting in Formal Language Theory*, volume 18 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1989.

[Eil74] Samuel Eilenberg. *Automata, Languages and Machines*, volume A of *Pure and Applied Mathematics*. Academic Press, 1974.

[ES90a] Susan Even and David A. Schmidt. Category-sorted algebra-based action semantics. *Theoretical Computer Science*, 77(1–2):73–95, December 1990.

[ES90b] Susan Even and David A. Schmidt. Type Inference for Action Semantics. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming (ESOP'90)*, volume 432 of *Lecture Notes in Computer Science*, pages 118–133, Copenhagen, May 1990. Springer-Verlag.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and it application to OOP. In *Proc. Mathematical Foundations of Programming Semantics*, 1995. To appear.

[FGM+97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *RFC-2068: Hypertext Transfer Protocol – HTTP/1.1.* IETF, January 1997.

[FH91] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10), October 1991.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley, 1995. ISBN 0-8053-1670-1, URL: `http://www.cs.princeton.edu/software/lcc/`.

[FM90] You-Chin Fuh and Prateek Mishra. Type Inference With Subtypes. *Theoretical Computer Science*, 73(1):155–175, 1990.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Fuc63] L. Fuchs. *Partially Ordered Algebraic Systems.* Intl. Series of Monographs in Pure and Applied Mathematics. Pergamon, 1963.

[GAK94] Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective λ-calculus. In *POPL'94: ACM Symp. on Principles of Programming Languages*, pages 35–47, Portland, OR, January 1994. ACM, ACM Press.

[GM82] J. Goguen and J. Meseguer. Security policies and security models. In *Proc. 1982 IEEE Symp. on Security and Privacy*, 1982.

[Grä78] George Grätzer. *General Lattice Theory*, volume 75 of *Pure and Applied Mathematics.* Academic Press, 1978.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[Har85] Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *STOC'85: Proc. of the 17'th Symp. on the Theory of Computing*, pages 185–194, Providence, May 1985. ACM, ACM Press.

[Hei93] Nevin Heintze. Set-based analysis of ML programs. Technical Report CMU-CS-93-193, CMU School of Computer Science, 1993. URL: ftp://reports.adm.cs.cmu.edu/usr/anon/1993/CMU-CS-93-193.ps.

[Hen92] Fritz Henglein. Dynamic typing. In *Proc. ESOP'92, European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 233–253. Springer-Verlag, 1992.

[HM94] Fritz Henglein and Christian Mossin. Polymorphic Binding-Time Analysis. In Donald Sannella, editor, *Proceedings of the 1994 European Symposium on Programming (ESOP'94)*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.

[HY86] Paul Hudak and Jonathan Young. Higher-order strictness analysis in untyped lambda-calculus. In *POPL'86: Proc. ACM Conf. on Principles of Programming Languages*, pages 97–108, St. Petersburg, FL, January 1986. ACM, ACM Press.

[Jak93] Tony Thisted Jakobsen. Towards a Realistic Type Inference System for Action Semantics. M.Sc. dissertation, Computer Science Department, University of Aarhus, Denmark, May 1993.

[JCØ94] Pierre Jouvelot, Charles Consel, and Peter Ørbæk. Separate Polyvariant Binding-time Reconstruction. Technical Report CRI-A/261, Ecole des Mines, Paris, July 1994. URL: `ftp://cri.ensmp.fr/pub/LOMAPS/LOMAPS-CRI-2.dvi.Z`.

[JM81] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In S. D. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice Hall, 1981.

[Jon81a] Neil D. Jones. Flow analysis and optimization of LISP-like structures. Technical Report PB-128, DAIMI, Comp. Sci. Dept. of Aarhus University, 1981.

[Jon81b] Neil D. Jones. Flow analysis of lambda expressions. In S. Even and O. Kariv, editors, *ICALP'81: Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 114–128. Springer-Verlag, July 1981.

[Jon87] Neil D. Jones. Flow analysis of lazy higher-order functional programs. In Samson Abramsky and Chris Hankin, editors, *Abstract interpretation of declarative languages*, chapter 5, pages 103–123. Ellis Horwood, 1987.

[KLMM93] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson. *Object Oriented Environments: The Mjølner Approach*. Prentice Hall, 1993. ISBN 0-13-009291-6.

[KR88] Brian W. Kernighan and Dennis M. Richie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[Las95] Søren B. Lassen. Basic action theory. Technical Report RS-96-26, BRICS, Comp. Sci. Dept., University of Aarhus, Denmark, May 1995. URL: `http://www.brics.dk/RS/95/25/BRICS-RS-95-25/BRICS-RS-95-25.html`.

[Las97] Søren B. Lassen. Action semantics reasoning about functional programs. *Mathematical Structures in Computer Science*, 1997. Special issue dedicated to the Workshop on Logic, Domains, and Programming Languages (Darmstadt, May 1995). To appear.

[Lau87] John Launchbury. Projections for specialisation. In Jones Bjørner, Ershov, editor, *Partial Evaluation and Mixed Computation: Proc. of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 299–315. IFIP, North-Holland, October 1987.

[LM69] E. S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12:13–22, 1969.

[LT79] Thomas Lengauer and Robert E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. on Programming Languages and Systems*, 1(1):121–141, 1979.

[Mit84] John Mitchell. Coercion and type inference. In *Eleventh Symposium on Principles of Programming Languages*, pages 175–185, 1984.

[Miz89] M. Mizuno. A least fixed point approach to interprocedural information flow control. In *Proc. 12th National Computer Security Conference*, 1989.

[MØ95] Karoline Malmkjær and Peter Ørbæk. Polyvariant Specialisation for Higher-Order, Block-Structured Languages. In William L. Scherlis, editor, *Proceedings of the 1995 ACM Conference on Partial Evaluation and Program Manipulation (PEPM'95)*, San Diego, June 1995. ACM, ACM Press.

[Mos92] Peter D. Mosses. *Action Semantics.* Cambridge University Press, 1992. Number 26 in the Cambridge Tracts in Theoretical Computer Science series.

[MS92] Masaaki Mizuno and David Schmidt. A security flow control algorithm and its denotational semantics correctness proof. *Formal Aspects of Computing*, 4(6a):727–754, 1992.

[MW94] Hermano Moura and David A. Watt. Action Transformations in the Actress Compiler Generator. In Peter Fritzon, editor, *Proceedings of the 1994 Conference on Compiler Construction, Edinburgh*, volume 786 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, April 1994.

[Nie87] Flemming Nielson. Strictness analysis and denotational abstract interpretation. In *POPL'87: Proc. ACM Conf. on Principles of Programming Languages*, pages 120–131, Munich, January 1987. ACM, ACM Press.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *POPL'96: Proc. of the 1996 ACM Conf. on Principles of Programming Languages*, pages 54–67, St. Petersburg, FL, January 1996. ACM.

[ØP97] Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(4), 1997.

[Ørb93] Peter Ørbæk. Analysis and Optimization of Actions. M.Sc. dissertation, Computer Science Department, University of Aarhus, Denmark, September 1993. URL: `ftp://ftp.daimi.aau.dk/pub/empl/poe/index.html`.

[Ørb94a] Peter Ørbæk. OASIS: An Optimizing Action-based Compiler Generator. In Peter Fritzon, editor, *CC'94: Proc. of the 1994 Conf. on Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 1–15, Edinburgh, April 1994. Springer-Verlag. URL: `ftp://ftp.daimi.aau.dk/pub/empl/poe/index.html`.

[Ørb94b] Peter Ørbæk. OASIS: An Optimizing Action-based Compiler Generator. In Peter D. Mosses, editor, *Proceedings of the First International Workshop on Action Semantics*, volume NS-94-1 of BRICS Notes Series, pages 99–112. BRICS, University of Aarhus, April 1994.

[Ørb95] Peter Ørbæk. Can you Trust your Data? In P. D. Mosses, editor, *Proceedings of the TAPSOFT/FASE'95 Conference*, volume 915 of *Lecture Notes in Computer Science*, pages 575–590, Aarhus, Denmark, May 1995. Springer-Verlag. URL: `ftp://ftp.daimi.aau.dk/pub/empl/poe/index.html`.

[Pal92] Jens Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Computer Science Department at University of Aarhus, January 1992.

[Pal95] Jens Palsberg. Efficient inference of object types. *Information and Computation*, 123(2):198–209, 1995. Preliminary version in Proc. LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science, pages 186–195, Paris, France, July 1994.

[PØ95a] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In Alan Mycroft, editor, *SAS'95: Static Analysis*, volume 983 of *Lecture Notes in Computer Science*, pages 314–330, Glasgow, September 1995. Springer-Verlag.

[PØ95b] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. Technical Report BRICS-RS-95-31, BRICS, University of Aarhus, June 1995.

[Pro59] R. T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *AFIPS Eastern Joint Computer Conf.*, pages 133–138, Baltimore, MD, 1959. Spartan Books.

[PS92] Jens Palsberg and Michael I. Schwartzbach. Binding Time Analysis: Abstract Interpretation vs. Type Inference. Technical Report PB-393, DAIMI, 1992.

[PS95] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *POPL'89: ACM Symp. on Principles of Programming Languages*, pages 77–87, Austin, TX, January 1989. ACM.

[Rém92] Didier Rémy. Typing record concatenation for free. In *POPL'92: Proc. Conf. on Principles of Programming Languages*, pages 166–176, Albuquerque, January 1992. ACM, ACM Press.

[Rey69] John C. Reynolds. Automatic computation of data set definitions. In A. J. H. Morell, editor, *Information Processing 68: Proceedings of the IFIP Congress 68*, volume 1, pages 456–461. North-Holland, 1969.

[Shi88] Olin Shivers. Control flow analysis in scheme. In *PLDI'88: Proc. SIGPLAN'88 Conf. on Programming Language Design and Implementation*, Atlanta, GA, June 1988. ACM. In SIGPLAN Notices vol. 23, no. 7.

[Shi91] Olin Shivers. Control-flow analysis of higher-order languages or taming lambda. Technical Report CMU-CS-91-145, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1991.

[Sta95] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 1995.

[Tip95] Frank Tip. *Generation of Program Analysis Tools*. Ph.d. thesis, CWI, University of Amsterdam, March 1995.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *LICS'87: IEEE Symposium on Logic in Computer Science*, pages 37–44. IEEE, 1987.

[Wan88] Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *LICS'88: IEEE Symposium on Logic in Computer Science*. IEEE, 1988.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *LICS'89: IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989. IEEE.

[WC94] Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. In *LFP'94: Proc. of the 1994 ACM Conf. on Lisp and Functional Programming*, pages 250–262. ACM, June 1994. URL: `http://www.neci.nj.-nec.com/homepages/wright.html`.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *POPL'94: Proc. of the 1994 ACM Symp. on Principles of Programming Languages*, pages 297–310. ACM, January 1994.

[Wei84] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, 1984.

[Wei92] Michael Weiss. The transitive closure of control dependence: The iterated join. *ACM Letters on Programming Languages and Systems*, 1(2):178–190, June 1992.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994. Preliminary version is Rice Tech. Report TR91-160.

[WH87] P. Wadler and R. J. M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *FPCA'87: Proc. of the 3rd Intl. Conf. on Functional Programming and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, Portland, OR, September 1987. ACM, Springer-Verlag.

[WS91] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, 1991.

Index

A_A, 16
abstraction, 140
Action, 139
Addr, 13
addr, 14
$addr_A$, 16
$addr_I$, 14, 15
$A_{i\cdot}$, 72
A_{ij}, 66
annihilator, 62
asg, 16
associative, 62

big, 55
Boolean, 139
boolean, 140

CDF(X, Y), 117
coalesced sum, 40
coherent model, 20
commutative, 62
concatenation, 72
Ct, 18
Curry type, 45

DAlg, 62
  structure, 62
decoration, 46
deductive closure, 54
dependence, 59, 60
  may-, 87
  must-, 87
dis, 42, 109
distributive, 62
domain
  of function, 17, 53
dom(M), 17

$\mathcal{E}$, 14
$\mathcal{E}_A$, 16
$\mathcal{E}_I$, 14
Env, 13, 40
Env_A, 16
erasure, 45
$\mathcal{E}_S$, 18

greatest lower bound, 62
grid, 81

Id, 147
idempotent, 62
I_k, 122
image, 70
  of sets of vectors, 75
img(V, L), 75
img(v, Q), 70
immediate post-dominator, 117
indep, 87, 109
instrumented semantics, 14
integer, 140
IPD(), 117

J_k, 121

K-grid, 81

language
  lambda calculus, 31
  While, 13
$L(C, X)$, 55
least upper bound, 62
limit, 63
$\overline{L}_{\mathsf{indep}}$, 87
L_{mn}, 83
L_T, 151

M_A, 16
may-dependence, 87
Mem, 13
Mem_I, 14
M_I, 14

M_{mn}, 66
$M_{mn}(D)$, 66
model
  coherent, 20
monoid, 63
must-dependence, 87

$n()$, 109
$n_2()$, 109
Natural, 139
non-terminal, 144
notation
  2^I, 16
  $\alpha_K(L)$, 82
  $\alpha_K^0(A)$, 76
  $\alpha_K^0(Q)$, 76
  $\alpha_K^0(v)$, 76
  $[]$, 109
  $P : [X \to Y]$, 116
  $a \cdot v$, 65
  $a \cdot b$, 62, 88
  $G \cdot G'$, 81
  $L \cdot L'$, 74
  $A \cdot B$, 66
  $Q \cdot Q'$, 68
  $M_I \circ A \sqsubseteq A_A$, 17
  $v + w$, 72
  $G + G'$, 82
  $L + L'$, 74
  $A + B$, 72
  $Q + Q'$, 72
  Δ, 46
  δ, 18
  $\delta(V)$, 55
  $A \vdash \langle v, t \rangle \sqsubseteq a$, 17
  $A \vdash_C E : t$, 45
  $A \vdash E : \tau$, 43
  η, 18
  $\gamma_K(G)$, 83
  $\sqcap$, 62, 140
  $\wedge$, 87
  $\leq$, 14, 87
  $\vee$, 87
  $\sqcup$, 62, 65, 66, 68, 88, 109, 140
  $m \models c$, 20
  $L, V' \models v, Q, w$, 152
  $S \models m, c$, 123
  $(\delta, \varphi) \models (C, D)$, 53
  $\oplus$, 40
  $\otimes$, 40
  (s, A, t, u), 67
  π_n, 15
  $\prod$, 92
  $\langle A, B \rangle$, 140
  $\langle \cdot, \cdot \rangle$, 15
  $\langle v, w \rangle$, 65
  $\langle GH \rangle_r$, 52
  Ψ, 91
  ρ, 40
  $[\![\cdot]\!]$, 40
  $[\![E]\!]_r$, 52
  $[\![E]\!]_y$, 52
  $\sigma[\![]\!]$, 140
  (s, m, p), 109
  $\sqsubseteq$, 61, 88, 109, 140
  $(A, B, v) \to (t, w)$, 139
  $(pt, qt, V, E) \triangleright (pt', qt', V', L)$, 150
  $(s, m, p) \mapsto (s', m', p')$, 109
  $(S, Q, p) \triangleright (S', Q', p')$, 123
  $(Y, B, v) \to w$, 139
  $f \subseteq g$, 150
  $+$, 62, 65, 66, 68, 88
  $t : s$, 109
  $\uplus$, 13
  $\mathbf{1}$, 62, 67
  $F[S_1 \mapsto S_2]$, 150
  $m[l \mapsto t]$, 109
  $M[x/a]$, 14
  v^t, 121
  $\mathbf{0}$, 62, 67, 68
nothing, 140
NT, 144
$\mathrm{nt}(V)$, 150

ordered semi-ring, 63
overlined element, 87

path, 116
post-dominator, 117
  immediate, 117
pre, 71
pre-image, 71
Primitive, 139
pt, 150

Q_{mn}, 68
$Q_{mn}(D)$, 68
qt, 150
Quad, 150
quad, 67
quad languages, 74
quadruple, 67

rev, 92
reverse image, 92

$\mathcal{S}$, 14
$\mathcal{S}_A$, 16
scalar product, 65
semantics
  instrumented, 14
  standard, 13
semi-ring, 63
  ordered, 63
sequential composition, 62, 66
$\mathcal{S}_I$, 14, 15
sink, 67
slot, 59
small, 55
smash product, 40
Sort, 139
src, 67
$\mathcal{S}_S$, 18
standard semantics, 13
sum, 62

T-system, 51
$T(E)$, 52
tensor, 65
Tr, 14
Tr, 109
tr, 42, 109
transpose, 121
typeof(v), 146, 152

Addr, 13
Val_A, 16
Val_I, 14
Var, 40
Vec, 150
$\mathcal{V}_r$, 51
$\mathcal{V}_y$, 51

While language, 13

$X_{\mathsf{completed}}[\![]\!]$, 147
$X_{\mathsf{escaped}}[\![]\!]$, 147
$X_{\mathsf{Y}}[\![]\!]$, 147

Yielder, 139

Recent BRICS Dissertation Series Publications

DS-97-2 Peter Ørbæk. ***Trust and Dependence Analyis*****. July 1997. x+175 pp.**

DS-97-1 Gerth Stølting Brodal. ***Worst Case Efficient Data Structures*****. January 1997. Ph.D. thesis. x+121 pp.**

DS-96-4 Torben Braüner. ***An Axiomatic Approach to Adequacy*****. November 1996. Ph.D. thesis. 168 pp.**

DS-96-3 Lars Arge. ***Efficient External-Memory Data Structures and Applications*****. August 1996. Ph.D. thesis. xii+169 pp.**

DS-96-2 Allan Cheng. ***Reasoning About Concurrent Computational Systems*****. August 1996. Ph.D. thesis. xiv+229 pp.**

DS-96-1 Urban Engberg. ***Reasoning in the Temporal Logic of Actions — The design and implementation of an interactive computer system*****. August 1996. Ph.D. thesis. xvi+222 pp.**