Flow Analysis of Typed Higher-Order Programs

(Revised Version)

Christian Mossin

DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø, Denmark

Technical Report DIKU-TR-97/1

A thesis submitted in partial fulfillment of the requirements for the degree of Ph.D. of University of Copenhagen

Abstract

This thesis concerns *flow analysis* of typed higher order programs. Flow analysis attempts at compile time to predict the creation, flow and use of values. We will present a suite of analyses based on type systems: they work by adding extra information to the standard types of the analysed program. The analyses are proven to be sound under any evaluation order.

The first two analyses, *simple* and *sub-type based* flow analysis, are known from the literature. We present a new *modular* formulation, allowing subexpressions to be analysed independently from their context without loss of precision.

Modularity makes it possible to extend the analyses with *polymorphism* which allows named functions to be used differently in different contexts. This leads to improved precision. We show that *ML*- and *fix*-polymorphic flow analysis is computable in polynomial time.

Finally, we present a flow analysis based on *intersection* types. We show a completeness result for this analysis: the analysis is precise up to not knowing which branch of a conditional is taken and not discarding any computation.

The sub-type based analysis can be given a very natural presentation using *graphs*. The graph formulation leads to an improvement over existing algorithms: single flow queries can be answered in linear time and full flow analysis can be performed in quadratic time (under assumption that the size of standard types is bounded).

We argue that the presented analyses are not restricted to a specific standard type system, but are applicable to any functional programming language; even dynamically typed languages.

Preface

This thesis is submitted for the degree of Ph.D. at DIKU, the department of computer science at the University of Copenhagen. It reports work done between September 1993 and December 1996 at DIKU and at University of Glasgow (where I was a guest from September 1994 to February 1995).

The contents of the thesis and the results presented are previously unpublished, though chapter 5 relies on work done in binding-time analysis with Fritz Henglein [HM94] and with Dirk Dussart and Fritz Henglein [DHM95a].

Acknowledgements

First of all, I would like to thank my supervisors Nils Andersen and Fritz Henglein for their invaluable help, support and inspiration.

My six month visit to Glasgow was very important to my personal as well as scholarly development. Phil Wadler and David N. Turner deserve thanks for making my stay very rewarding and pleasant. The atmosphere at the department was very amiable and I felt quickly at home.

During my studies, I have paid short visits to a number of institutions. These have all proven to be inspirational and rewarding. Thanks go to Dirk Dussart and Karel de Vlaminck (Leuven), Alex Aiken (Berkeley), Kathleen Fisher (Stanford), Patrick Lincoln (CRI) and Hanne and Flemming Nielson (DAIMI).

At times when I have been stuck, I sought help via e-mail. The following persons answered my (sometimes naive) questions and engaged in discussions that gave me the understanding and intuition I sought: Mariangiola Dezani, Thomas Jensen, Ian Mackie and Pawel Urzyczyn.

DIKU is a great and inspirational place to do your thesis: people are always willing to discuss your problems and often leave you with a better understanding. With the risk of unjust omissions, I would like to mention Jesper Jørgensen, Jakob Rehof and Morten Heine Sørensen for many good and rewarding discussions. How could you manage the solitary thesis work without inspiring office mates: Jesper Jørgensen, Kristoffer Rose (his XY-pic was used for the drawings [HR95]), Andrew Partridge and Peter Holst

Andersen.

Finally, I would like to thank my understanding girlfriend Tina for her support and encouragement.

The work was made possible by a grant from the Danish Technical Research Council (phd-stipendium) and by employment at DIKU (forskningsassistent).

Revision

The present report is a revised version of the thesis successfully defended January 31st 1997. This version incorporates the improvements suggested by the thesis committee — I heartily thank Alex Aiken, Nils Andersen and Peter Sestoft for their thorough work on commenting this work. Furthermore, the revision contains some improvements and corrections discovered while pursuing the ideas of the thesis further ([Mos97b, Mos97a]). In particular, conjecture 6.8 of the original version has been proven (theorem 6.22) and the suggestion for handling recursive types in flow graphs (section 9.3.1 in both versions) which turned out to be unsound has been corrected.

Copenhagen, August 1997

Contents

1 Introduction **1**
- 1.1 Program Analysis . . . 1
- 1.2 What is Flow Analysis? . . . 2
- 1.3 Analysing Typed Programs . . . 3
  - 1.3.1 Type Based Analysis . . . 4
  - 1.3.2 Graph Based Analysis . . . 5
- 1.4 This Thesis . . . 6
- 1.5 Outline . . . 7
- 1.6 Language . . . 10
- 1.7 Flow Analysis . . . 12
  - 1.7.1 Flow Properties . . . 13
  - 1.7.2 Flow Functions . . . 14

I Monovariant Analysis **17**

2 Simple Types **19**
- 2.1 The Type System . . . 19
  - 2.1.1 Simple Flow Functions . . . 21
- 2.2 Principality and Minimality . . . 21
  - 2.2.1 Principality . . . 22
  - 2.2.2 Minimality . . . 28
- 2.3 Soundness . . . 29
- 2.4 Constraint Based Analysis . . . 32
  - 2.4.1 Equivalence between Simple Flow Analysis and CFA via Equality . . . 34
- 2.5 Algorithm . . . 35

3 Subtyping **37**
- 3.1 Subtyping . . . 37
  - 3.1.1 Subtyping Flow Functions . . . 39
- 3.2 Principality and Minimality . . . 39
  - 3.2.1 A Syntax Directed System . . . 42

3.2.2 Algorithm . . . 42
3.2.3 Minimality . . . 50
3.3 Sestoft's Closure Analysis . . . 52
3.4 Constraint Based Analysis . . . 53
3.5 Equivalences . . . 54
3.5.1 Equivalence between Constraint Based Analysis and Closure Analysis . . . 54
3.5.2 Equivalence between Subtype and Constraint Based Analysis . . . 57
3.5.3 Complexity . . . 58
3.6 Soundness . . . 58

4 Flow Graphs **61**
4.1 Untyped graphs . . . 62
4.1.1 Pre-flow-graphs . . . 63
4.1.2 Closing Pre-flow-graphs . . . 66
4.1.3 Equivalence to Constraint Based Analysis . . . 68
4.2 Typed graphs . . . 68
4.3 Typed and Untyped Graphs . . . 72
4.3.1 Paths in untyped graphs are in typed graphs . . . 73
4.3.2 Paths in typed graphs are in untyped graphs . . . 74
4.4 Modularity and Algorithms . . . 76
4.5 Paths by Asperti and Laneve . . . 77
4.6 Summary of Monovariant Analyses . . . 81

II Polyvariant Analysis **83**

5 Polymorphism **85**
5.1 Polymorphic Formulae and Logical Rules . . . 85
5.2 ML polymorphism . . . 87
5.2.1 Syntax Directed Type System . . . 87
5.2.2 What is the Result of Polymorphic Flow Analysis . . . 89
5.2.3 Halbstark Instance Relation . . . 92
5.2.4 Algorithm and Principality . . . 97
5.2.5 Accelerated Algorithm and Minimality . . . 101
5.2.6 Complexity . . . 104
5.2.7 Soundness . . . 105
5.2.8 Invariance under Transformation . . . 107
5.3 Polymorphic recursion . . . 110
5.3.1 Syntax Directed System . . . 111
5.3.2 Polymorphic Recursion Revisited: Kleene-Mycroft Iteration . . . 112
5.3.3 Algorithm I: Principality . . . 114

5.3.4 Algorithm II: Bounding Kleene-Mycroft Sequences . . 122
5.3.5 Algorithm III: Avoiding Recomputation 124
5.3.6 Complexity 125
5.3.7 Soundness 127
5.3.8 Invariance under Transformation 128

6 Intersection Types **133**
6.1 Interpreting Intersection Type Derivations 135
6.2 Decidability 138
6.3 Minimality 145
6.4 Non-Standard Semantics 147
6.4.1 Values 151
6.5 Subject Reduction 152
6.6 Subject Expansion 154
6.7 Handling 'fix' 159
6.8 Flow in Normal Forms 162
6.9 Summarising the Results 165

7 Shivers' CFA **167**
7.1 0CFA 167
7.1.1 Pairs 171
7.2 nCFA 172
7.3 CPS vs. direct style 173

III Extensions and Applications **177**

8 Extensions **179**
8.1 Reachability 179
8.1.1 Reachability and Intersection Flow Analysis 180
8.1.2 Reachability in Graphs 182
8.2 Union Types 182
8.3 Usedness 185
8.3.1 Lazy Evaluation 185
8.3.2 Eager Evaluation 186
8.4 Simple Polymorphism 186
8.5 Other Inference-Based Analyses 187
8.6 Labelled Graphs 187
8.7 Shivers' CFA and Linearity 190

9 Other Standard Type Systems **193**
9.1 ML Polymorphism 193
9.1.1 ML Polymorphism in Typed Graphs 194
9.2 Sum Types 197

9.2.1 Sum Types in Typed Graphs . . . 197
9.3 Recursive Types . . . 200
9.3.1 Recursive Types in Typed Graphs . . . 201
9.4 Dynamic Types . . . 202
9.4.1 Dynamic Types in Typed Graphs . . . 205

10 Applications **209**
10.1 Constant Propagation . . . 209
10.1.1 Simple and Subtype Constant Propagation . . . 210
10.1.2 Polymorphic Constant Propagation . . . 210
10.1.3 Intersection Constant Propagation . . . 212
10.1.4 Graph Based Constant Propagation . . . 212
10.2 Firstification . . . 212
10.3 Binding-Time Analysis . . . 214
10.3.1 Type-Based BTA . . . 214
10.3.2 Graph-based BTA . . . 218

11 Related Work **221**
11.1 Abstract Interpretation . . . 221
11.2 Constraint Based Analysis . . . 223
11.3 Set-Based Analysis . . . 224
11.4 Type Based Analysis . . . 225
11.5 Safety, Type Recovery and Soft Typing . . . 226

12 Conclusion **229**
12.1 Summary . . . 229
12.2 Future Work . . . 230
12.2.1 Robustness . . . 230
12.2.2 Making the Analyses more Generally Applicable . . . 230
12.2.3 Improvements of the Analyses . . . 230

Dansk sammenfatning **231**

Bibliography **232**

Index **240**

List of Figures

1.1 Type system 11
1.2 Logic of Properties 14

2.1 Simple flow analysis — formulae 19
2.2 Simple flow analysis — non logical rules 20
2.3 Algorithm $\mathcal{W}$ for simple types 23
2.4 Constraint generation a la Heintze 33

3.1 Subtyping flow analysis — formulae and subtype relation . . 38
3.2 Subtyping flow analysis — non-logical rules 40
3.3 Syntax-directed subtyping flow analysis 41
3.4 Algorithm $\mathcal{W}$ 44
3.5 Constraint generation a la Palsberg 54

4.1 Pre-flow-graphs (1) 63
4.2 Pre-flow-graphs (2) 64
4.3 Closing rules 67
4.4 Typed flow graphs (1) 70
4.5 Typed flow graphs (2) 71

5.1 Polymorphic flow analysis — formulae and subtype relation . 86
5.2 ML-polymorphic flow analysis — non-logical rules 88
5.3 Syntax directed ML-polymorphic flow analysis 89
5.4 ML-polymorphic flow function 91
5.5 Algorithm $\mathcal{W}$ for polymorphic recursion 126

6.1 Intersection flow analysis — formulae 134
6.2 Intersection flow analysis — logical rules 135
6.3 Intersection flow analysis — type-specific rules 136
6.4 Intersection flow analysis — non-logical rules 137
6.5 Intersection flow analysis — semi logical rules 137
6.6 Syntax Directed Intersection Flow Analysis 140
6.7 Non-standard reduction 149
6.8 Non-standard reduction — context rules 150

7.1 0CFA 168
7.2 0CFA for a tail-recursive language 174

9.1 ML-polymorphism 194
9.2 Flow analysis of ML-polymorphic programs 195
9.3 Sum Types 197
9.4 Flow analysis with sums 198
9.5 Typed flow graphs for sum types 199
9.6 Fold and unfold 203
9.7 Dynamic Typing 204
9.8 Flow analysis and dynamic typing 206
9.9 The $[Fun!]^l$ m-node 207

Chapter 1

Introduction

In the childhood of automatic computing, programs were small and written in the machine code of the machine on which the program was intended to run. Programming languages facilitated portability between different machines. Even more important, they allowed the programmer to abstract from details and leave these to the compiler.

With the growth of computers (in size and speed) abstract programming languages were no longer a convenience but a prerequisite for program-development. A more recent benefit was the ability to restrict the direct access from a program to storage and external devices; this allows safe execution of (possibly unknown) programs.

One programming language paradigm that seems to be reaching maturity these years is functional programming. This allows high-level abstract specification of programs based on sound mathematical principles. In strongly typed variants of functional programming a high level of safety is implied.

1.1 Program Analysis

The more abstract programming languages get, the more they distance themselves from the underlying machine, the more will be required from compilers in order to produce efficient code. The point of abstract programming languages was exactly to allow the programmer to leave out details that were not important for the specification of the solution. Those details might, however, be crucial to the execution of the program.

Program analyses allow the compiler to infer some of the information that was left out by the programmer. A program analysis usually goes hand in hand with a program transformation that will optimise the program based on the information inferred by the analysis. There are some common prerequisites for most program analyses:

- They require information about the flow of *data* within the program.

- They require information about the possible *control* paths through the program.

The study of *data-flow* analysis and *control-flow* analysis allows a general perspective on program analyses and allows the study of generally applicable algorithms. Furthermore the result of data- and control-flow analysis can be used directly as the basis for other analyses.

1.2 What is Flow Analysis?

In imperative languages we have:

Data flow analysis: "...traces the possible definitions and uses of data in the program"

Control flow analysis: "...traces the patterns of possible execution paths in a program"

(Quotations from [RP86]). For higher order languages we cannot separate the two: consider the following expression:

$$\begin{array}{l}\text{let } f = \lambda g.g\underline{@}\text{True} \\ \text{ in } f@\lambda x.\text{if } x \text{ then False else True}\end{array}$$

Data flow analysis could infer that the value 'λx.if x then False else True' can flow (through g) to the underlined application. But the fact that control at this application passes to the body of the let-bound λ, can be considered *control-flow*. Only due to inferring this pass of control are we able to infer that the value True can flow (through x) to the conditional of the 'if' (again data-flow).

As we see, there is an interdependency between data-flow and control-flow analysis. This naturally leads to the conclusion that the two should be combined into one analysis. The combined analysis has previously been referred to as control-flow analysis (Shivers), flow analysis (Jaganathan, Weeks, Wright and Ashley) and closure analysis (where only flow of functions (and sometimes data-structures) is taken into consideration; Sestoft, Bondorf). The general term "flow analysis" seems appropriate for the analyses presented later in this thesis as they will contain elements from both data- and control-flow analysis.

There is, however, an aspect of control-flow analysis that is *not* captured by the analyses above: the order of evaluation. For imperative languages this is part of the result of a control-flow analysis. The reason is that evaluation order is usually an integral part of the definition of an imperative language. For a given functional language this is of course also the case, but there has been a tradition in the functional programming community

to do theoretical work on the lambda calculus with beta-reduction as semantics without specifying the order of evaluation. This makes analyses applicable to any functional language. Furthermore, it allows compile-time optimisations based on beta-reduction and it allows parallel execution of the program. On the other hand, it can restrict the precision of analyses. Based on this discussion, we might find it more appropriate to coin the previously designed analyses as data- or value-flow analyses.

Reduction-order independent flow analysis allows validity under any reduction order but prevents the analysis from being the basis for analysis that *does* rely on flow-dependencies. We will see how restricting ourselves to reduction-order independent analyses implies that certain versions of binding-time analysis cannot be based directly on flow analysis.

The analyses presented will follow this general tradition of not specifying evaluation order. We will, however, propose a few possible optimisations based on knowledge of evaluation order.

1.3 Analysing Typed Programs

The tradition of program analysis (with certain exceptions) has been to analyse the program itself, independent of whether the analysis is applied to typed or untyped programs. This, of course, allows a common specification of analyses for typed and untyped languages. On the other hand, if type inference is performed, the result of the type inference contains a lot of additional information about the program.

This intuition was explored from a different angle by Palsberg and O'Keefe, who investigated exactly which type information could be immediately inferred from flow information [PO95], and Heintze, who also investigated the other direction: which flow information is present in type derivations [Hei95].

The approach taken in this thesis is to assume that a well-typed program is given *along with the type information.* In other words, we assume that we are given an explicitly typed term. As noted above this gives us a better starting point for doing program analysis, but it also gives a natural separation of the problem: it is easily shown that the size of an explicitly typed term can potentially be exponential in the size of the term itself. This implies that all analyses presented in this thesis will have exponential worst case behaviour in the untyped term. On the other hand, this is an exponential factor which we know and love: in practice we can assume that the typed program is either given by the programmer (as in Pascal, C etc.) or is not much bigger than an underlying program for which type inference is performed (ML, Haskell etc.).

The idea is that part of the complexity of some analyses is really due to an attempt to reconstruct the standard type information during analysis.

Since type inference will be performed anyway for typed languages, and since this is known to be well-behaved in practice (despite its seemingly prohibitive theoretical worst-case behaviour), we can hope that, if we factor out the type-inference part of the analysis, the "remaining" complexity is much lower.

An example of this is *closure analysis* which is shown to have cubic complexity. This was believed to be a tight bound, but we will give an algorithm which is quadratic under assumption that all types are of bounded size — hence the complexity of our analysis is exponential in the size of the untyped program, but better behaved than closure analysis on "real" programs. Furthermore, our analysis allows single queries in linear time, in contrast to traditional closure analysis, which also required cubic time for this.

This approach implies that we need to fix not only the language, but also the type system: we will call such types *standard* types. We will choose the simplest type system possible: simple types. We will later show that we can extend the type system orthogonally to the choices made when specifying a given analysis. If a type system gives less than exact information about the type of a given subexpression, the result of the analysis will also be less precise: eg. *recursive* types allow infinite types to be given a finite description provided that the infinite structure in a sense "repeats" itself. This regularity will be conveyed to the flow analysis where repetition of flow results will be enforced as well.

By choosing a sufficiently liberal standard type system such as *dynamic* types, we can even analyse untyped programs. Thus, our methods do not restrict the choice of programming language; they only require that *some* type-discipline is enforced previously to analysis. The choice of type-discipline is important to the precision of the analysis.

1.3.1 Type Based Analysis

Type based analysis works by *annotating* or *refining* standard types by adding extra information to each type constructor in (standard) type inference trees. Such annotated types will be called *properties* or *flow types*. The annotations are more precise information about the term possessing the type which is being annotated. When doing flow analysis, annotations will consist of information about *which* value the expression can evaluate to. In a judgement e : Bool, the type describes the sort of value that e can evaluate to while an annotated type Bool^L will state that e can evaluate to one of the values described by L (which all will be of type Bool).

Type based analysis allows a natural separation of specification and implementation of an analysis. There is no method of computation implied by the formalism. The formalism specifies analysis in a *local*, *relational* manner that allows for easy reasoning about properties of the program.

Compared to abstract interpretation (in the style of the Cousots [CC77]), one might say that abstract interpretation *combines* the issues of specification, implementation and correctness, while type based analysis *separates* these issues. Soundness is usually built into the construction of an abstract interpretation based analysis, but it can be difficult to prove other properties about the analysis. Finally, if an analysis is obtained from a (possibly instrumented) denotational semantics, this semantics have already made a choice concerning evaluation order. In this case it can be difficult to specify reduction-order independent analyses.

Abstract interpretation and type-based analysis are not mutually exclusive methods, a standard type can, after all, be seen as an abstraction of a big-step operational semantics. We do, however, believe that certain analyses are more easily defined in one framework than in the other. One big advantage of the type-based approach is that concepts and results from standard type theory often can be adopted to program analysis with little change.

Type based analysis usually adds annotations directly to the standard types without changing the structure. It is, however, possible to refine the structure as well:

1. By adding *polymorphism* over annotations. This polymorphism is completely separate from the standard type system (which might contain standard polymorphism over types). An example of this is Dussart, Henglein and the present author's work on polymorphic binding-time analysis [HM94, DHM95a].

2. Adding new connectives such as intersection and union. This requires that the underlying type of each component of the intersection or union is identical (but naturally allows the annotations to differ). An example of this can be found in Jensen's work on strictness analysis [Jen92].

Type based program analysis can take advantage of the "implicit" flow information contained in the type derivation. It does, however, also inherit the potential problem of exponential behaviour. This, as discussed above, should not cause any grief.

1.3.2 Graph Based Analysis

The result of data-flow analysis for imperative programs is usually represented by a *flow graph* which can be inferred directly from the program text. Using a graphical representation of constraint based analysis, we arrive at a very natural form of higher-order flow graph: the basic flow of simple constraints forms a *pre-flow graph* (which resemble the syntax tree

for the term) and the induced flow of conditional constraints becomes *closing rules* on the graph.

The flow of a value is represented by paths from the value in the graph. We are thus able to trace the flow values through variables to the uses of the value.

Such graphs inherit the problems of constraint based flow analysis: the analysis is global and the implementation becomes cubic. We therefore introduce the concept of a *typed graph*.

There is a path between two nodes in a typed graph if and only if there is a path between the corresponding nodes in the graphs described above (untyped) and hence the precision of an analysis based on typed graphs is the same.

The advantage of typed graphs is that they can be generated by a single pass over the standard type derivation — the need for closing rules is avoided. This allows graphs to be constructed for individual modules and by attaching dummy values to input edges a modular analysis can be achieved.

The typed graphs will have a size equivalent to the size of the basic expression with explicit types on all subexpressions. If we assume all types to have bounded size, the complexity of flow analysis is improved over previously known analyses: it can be computed by simple reachability in quadratic time. Furthermore, the formalism allows single queries (such as "which values are consumed at this program point?" or "where is this value consumed?") to be computed in linear time.

1.4 This Thesis

The goal of this thesis is to investigate flow analysis of typed higher-order functional programs. In the development of analyses we will emphasize the following:

- **Intensional behaviour:** The result of flow analysis is not just a description of the behaviour of the whole program, but also a description of the internal behaviour. In particular, an analysis is sound if the flow description of *every* subexpression is preserved by reduction (except the redex itself) — it is insufficient to consider the description of the whole program. Many analyses described in the literature are presented in an *extensional* manner, such that the result of the analyses is a global description of the behaviour of the whole program.

- **Modularity:** It is inconvenient or even prohibitive for program development if the whole program has to be recompiled every time a change has been made. We therefore find it important that modules can be analysed *separately*. Separate analysis of modules should not result in loss of precision of the analysis: the result should be the same as if the

whole program was analysed at once. Until recently, the value-flow analysis for higher-order programs described in the literature were global. One exception is the work by Tang and Jouvelot, who use a type-based interface between different modules [TJ94]. Their approach, however, gives less precise results when analysing modules individually than if the whole program was analysed.

Principality as known from standard type inference and other kinds of program analyses yields separate analysis without loss of precision. We will prove this property for our type based analyses. Independently of this work, Flanagan and Felleisen recently used a similar idea to achieve separate set-based analysis [FF96].

- **Practicality:** We will strive for practical analysis. In particular, the complexity of analyses presented should not be forbidding for practical use. We give a novel formalism for closure analysis [Ses88] which improves the best known complexity of the analysis (independently of this work, the same result was discovered using a different approach by Heintze and McAllester [HM97]). We present two new polymorphic analyses, which improve the precision of closure analysis, and prove the analysis to have polynomial complexity.

- **Evaluation-order independence:** The developed analyses should be sound under any reduction order. This makes the analyses widely applicable, and perhaps even more important, does not prohibit compiler optimisations which do not adhere to the evaluation order of the language (eg. constant propagation is based on "deep" beta-reduction and might invalidate the result of an analysis mimicking outermost reduction).

An additional contribution of the thesis is *intersection* based flow analysis. Though this analysis is not practically applicable, its formulation and properties are theoretically appealing and we believe that it can serve as a good starting point in the search for more precise yet practical analyses. We give an exact characterisation of the precision of the analysis which shows that the analysis only errs by assuming that no reductions are ever discarded. This result implies that the analysis must be non-elementary recursive.

1.5 Outline

The rest of this chapter defines the basic simply typed language we will be using and gives a number of basic definitions concerning the nature of flow analysis.

Part I concerns *monovariant* analyses. By monovariance we mean that every definition is given *one* description which has to suffice for all contexts in which the defined expression is used.

Chapter 2 presents a very simple type based flow analysis. The analysis is very crude but can be implemented very efficiently. It will also serve as a first introduction to type based program analysis introducing central concepts in a simple framework. The analysis allows the same precision as analyses described by Bondorf and Jørgensen [BJ93] and Heintze [Hei95]. In contrast to these analyses, our formulation

- allows an intensional soundness proof: we show that the inferred flow information is preserved for every subexpression in the program.
- is modular: we prove that the analysis has principal typings. That is, for every expression, we can find a single flow description that without loss of precision can be used in any context. Furthermore, we identify a minimal, principal typing where all flow, that can be completely resolved without knowledge of the context, is resolved. The proof of existence of principal and minimal, principal typings is constructive.

Chapter 3 introduces subtypes in the analysis of chapter 2. This results in an analysis of the same strength as closure analysis [Ses88] and constraint based analysis [Pal94] both of which are presented. In contrast to these analyses, our analysis enjoys the same properties as simple type based flow analysis: intensional soundness and (constructive) existence of minimal, principal types.

Chapter 4 introduces flow graphs of equivalent accuracy as the analysis presented in chapter 3: *untyped graphs* are applicable to untyped languages and can be seen as a graphical representation of constraint based analysis, whereas *typed graphs* make constructive use of the available type information to achieve an improvement of the previously best known complexity of these analyses. We relate the paths in flow graphs to well-balanced paths known from optimal reduction. The definition of optimal reduction is that no redex is ever copied: this definition presupposes that the redexes that can occur during reduction of a term can be inferred from the term — essentially a flow analysis.

Part II concerns *polyvariant* analyses. Polyvariance allows several descriptions for every definition. This can be achieved by reanalysing the definition for every context in which it is used. This will however be prohibitively complex (or even undecidable in the case of recursive definitions) so the goal is to find a formalism that allows a uniform representation of these descriptions. We also wish the formalism to allow modularity such that the definition can be analysed independently of the contexts in which it will be used.

Chapter 5 combines the subtypes of chapter 3 with polymorphism. The introduction of polymorphism is made possible by the existence of principal types in the subtype based flow analysis. We present let- and fix-polymorphic analyses which allow definitions to be polymorphic: the addi-

tional precision achieved is characterised by showing that the analysis *cannot* be improved by let-unfolding resp. fix-unrolling. We show that the analyses have minimal, principal typings and present polynomial time algorithms.

Chapter 6 extends the subtype based analysis with intersection types. These give a strictly more precise analysis than polymorphism. We give a semantic characterisation of the strength of the analysis: we introduce a non-standard reduction system that chooses both branches of conditionals and never throw away redexes. With this system, the analysis is shown to be invariant under reduction and expansion.

Chapter 7 reviews Shivers' nCFA flow analysis [Shi91c]. OCFA has often been confused with the analyses presented in chapters 3 and 4 but we show it to be fundamentally different (0CFA is strictly stronger than closure analysis).

The last part (part III) discusses extensions of the analysis, both w.r.t. achieving better results and making them applicable to languages with other type systems. We go on to present applications of the analyses. This part is less verbose than the previous — the aim is to convince the reader that the analyses are useful and not restricted to the simply typed language used through the first sections. Finally, we discuss related work and conclude.

Chapter 8 discusses improvements of the analyses. We examine how to lift the assumption that both branches of conditionals always can be taken, and discuss how this relates to adding *union* flow types to our analyses. It is discussed how to take advantage of knowledge about the evaluation order: certain redexes will never be reduced under a fixed evaluation strategy (e.g. call-by-need). We also discuss various restrictions and extensions of polymorphic and intersection based flow analysis.

We present an idea for an improvement of graph based analysis inspired by work for flow analysis for imperative languages by Horowitz, Reps and Sagiv [RHS95, RSH94, SRH95]. Finally, we present an improvement of Shivers' nCFA based on usage information.

Chapter 9 shows how the ideas and algorithms presented in this thesis generalise to more complex languages than the simple one used for presentation. In particular, we discuss extensions of the type system with ML polymorphism, sum types, recursive types and dynamic types. The latter allows our analyses to be applied to untyped programs. The extensions are described for both type based and graph based analyses.

Chapter 10 discusses applications of flow analysis. We will focus on constant propagation, firstification and binding-time analysis. This illustrates the strength and applicability of our analyses, but also exposes the weaknesses inherent from deciding to make our analyses evaluation-order independent: certain versions of binding-time analysis cannot be based directly on flow analysis.

Chapter 11 discusses related work and compares. Chapter 12 concludes and discusses future work. The thesis ends with a summary in Danish.

The analyses presented in chapters 2, 3 and 4 are well known, though the presentation differs somewhat from earlier presentations of the same analyses (the graph representation is new). Some results are new, in particular principality and the connection to well-balanced paths of optimal reduction are to our knowledge new.

The analyses of chapters 5 and 6 as well as the more speculative chapter 8 present analyses which are all previously unknown. The polymorphic analysis of chapter 5 builds on foundations laid in binding-time analysis by Dussart, Henglein and the present author, while the intersection-based analysis is previously unpublished.

1.6 Language

This section defines the language which we will be analysing in the rest of the thesis (except chapter 9 where we will discuss extensions of the language). Let V be an enumerable set of variables and let $x, y, z, \ldots$ range over V. The set Exp_{pseudo} of *pseudo-terms* is given by the abstract syntax (where $e, e', e'', e_1, e_2, \ldots$ range over Exp_{pseudo})

$$e \quad ::= \quad x \mid \lambda x.e \mid e@e' \mid \mathrm{fix}\, x.e \mid (e, e') \mid \mathrm{let}\ (x, y)\ \mathrm{be}\ e\ \mathrm{in}\ e' \mid \mathrm{let}\ x = e\ \mathrm{in}\ e' \mid \mathrm{if}\ e\ \mathrm{then}\ e'\ \mathrm{else}\ e'' \mid \mathrm{True} \mid \mathrm{False}$$

The set T of types[1] is defined by:

$$t \quad ::= \quad \mathrm{Bool} \mid t \times t \mid t \to t$$

where $t, t', t'', t_1, t_2, \ldots$ range over T.

An occurrence of a type constructor Bool, $\to$ or $\times$ can be *positive* or *negative* in a type t:

1. Bool is a positive occurrence in Bool, $\to$ is a positive occurrence in $t \to t'$ and $\times$ is a positive occurrence in $t \times t'$.

2. If a type constructor c is a positive occurrence in t then it is a negative occurrence in $t \to t'$ and a positive occurrence in $t' \to t$, $t \times t'$ and $t' \times t$.

3. If a type constructor c is a negative occurrence in t then it is a positive occurrence in $t \to t'$ and a negative occurrence in $t' \to t$, $t \times t'$ and $t' \times t$.

If $A : V \to T$ is a partial map from variables to types, derivable type judgements $A \vdash e : t$ define a relation over $(V \to T) \times \mathrm{Exp}_{pseudo} \times T$. We read

[1] We will refer to these types as *standard types* to distinguish them from *annotated types* as introduced in following chapters

$A \vdash e : t$ as "e has type t under assumptions A". The set $\mathrm{Exp} \subseteq \mathrm{Exp}_{pseudo}$ of *expressions* or *terms* denotes the set of pseudo expressions e for which there exists A, t such that $A \vdash e : t$ is provable in figure 1.1.[2]

We write $A, x : t$ for the function A' such that

$$A'(y) = \begin{cases} t & , \text{if } y = x \\ A(y) & , \text{otherwise} \end{cases}$$

$$\text{Id}\ \frac{}{A, x : t \vdash x : t}$$

$$\rightarrow\text{-intro}\ \frac{A, x : t \vdash e : t'}{A \vdash \lambda x.e : t \rightarrow t'} \qquad \rightarrow\text{-elim}\ \frac{A \vdash e : t' \rightarrow t \quad A \vdash e' : t'}{A \vdash e@e' : t}$$

$$\text{Bool-intro}\ \frac{}{A \vdash \mathrm{True} : \mathrm{Bool}} \qquad \frac{}{A \vdash \mathrm{False} : \mathrm{Bool}}$$

$$\text{Bool-elim}\ \frac{A \vdash e : \mathrm{Bool} \quad A \vdash e' : t \quad A \vdash e'' : t}{A \vdash \text{if } e \text{ then } e' \text{ else } e'' : t}$$

$$\times\text{-intro}\ \frac{A \vdash e : t \quad A \vdash e' : t'}{A \vdash (e, e') : t \times t'} \qquad \times\text{-elim}\ \frac{A \vdash e : t \times t' \quad A, x : t, y : t' \vdash e' : t''}{A \vdash \text{let } (x, y) \text{ be } e \text{ in } e' : t''}$$

$$\text{fix}\ \frac{A, x : t \vdash e : t}{A \vdash \text{fix}\, x.e : t} \qquad \text{let}\ \frac{A \vdash e : t \quad A, x : t \vdash e' : t'}{A \vdash \text{let } x = e \text{ in } e' : t'}$$

Figure 1.1: Type system

Let e be an expression. Then $FV(e)$ and $BV(e)$ are the sets of *free* resp. *bound* variables of e defined as usual. We call e *well-named* if all free and bound variables of e are distinct. For every expression e there exists an α-equivalent expression e' such that e' is well-named. Unless stated otherwise, whenever we refer to an expression or term it will be assumed to be well-named.

A *context* is a term with one hole:

$$\begin{aligned} C \quad ::= \quad & [\,] \mid \lambda x.C \mid C@e \mid e@C \mid \text{fix}\, x.C \mid \text{let } x = C \text{ in } e \mid \text{let } x = e \text{ in } C \mid \\ & \text{if } C \text{ then } e' \text{ else } e'' \mid \text{if } e \text{ then } C \text{ else } e'' \mid \text{if } e \text{ then } e' \text{ else } C \mid \\ & (C, e') \mid (e, C) \mid \text{let } (x, y) \text{ be } C \text{ in } e' \mid \text{let } (x, y) \text{ be } e \text{ in } C \end{aligned}$$

We write $C[e]$ for the term obtained by replacing $[\,]$ in C with e. We assume that the resulting term is well-named.

[2] As we consider types an integral part of the definition of expressions (Church-style), we will always assume that the type of an expression is available.

If e is an expression with n occurrences of a variable x, we use $x^{(i)}$ to denote the i'th occurrence of x in e.

A *substitution* on terms is a map from V to Exp. A substitution from x to e is written $[e/x]$ and is applied to an expression e' using post-fix notation. If there are n occurrences of x in e we define $e[e'/x]$ to be the result replacing each occurrence $x^{(i)}$ by e'_i where

1. $e'_i =_\alpha e'$ and e'_i is well-named,
2. $BV(e'_i) \cap BV(e'_j) = \emptyset$ for $i \neq j$, and
3. $BV(e'_i) \cap (FV(e) \cup BV(e)) = \emptyset$

If $BV(e') \cap (FV(e) \cup BV(e)) = \emptyset$ we can safely assume that $e'_1 = e'$ without α-conversion.

The semantics of the language is defined by the following reductions:

(β)	$(\lambda x.e)@e'$	$\longrightarrow$	$e[e'/x]$	
(δ-if)	if True then e else e'	$\longrightarrow$	e	
	if False then e else e'	$\longrightarrow$	e'	
(δ-let)	let $x = e$ in e'	$\longrightarrow$	$e'[e/x]$	
(δ-let-pair)	let (x, y) be (e, e') in e''	$\longrightarrow$	$e''[e/x][e'/y]$	
(δ-fix)	fix $x.e$	$\longrightarrow$	$e[\text{fix}\, x.e/x]$	
(Context)	$C[e]$	$\longrightarrow$	$C[e']$	if $e \longrightarrow e'$

As usual we write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$

Note that reduction preserves well-namedness: if e is well-named and $e \longrightarrow e'$ then e' is well-named.

1.7 Flow Analysis

Let $\mathcal{L}$ be an enumerable set of *labels*. We let $l, l_1, l_2 \ldots$ range over labels. Given a program e a *labelling* is a function $\text{LabelOf}_e : \text{Exp} \to \mathcal{L}$ mapping occurrences of expressions to labels[3]. We use $\text{ExpOf}_e : \mathcal{L} \to \mathcal{P}(\text{Exp})$ for the function finding the set of sub-expressions of e with a given label. Thus

$$\text{ExpOf}_e(\text{LabelOf}_e(e')) \supseteq \{e'\}$$

and

$$\forall e' \in \text{ExpOf}_e(l) : \text{LabelOf}_e(e') = l$$

Let $\mathcal{L}_e$ denote the range of LabelOf_e.

[3]We could have required labelings to be injective (such that for each e', e'' subexpressions of e, $e' \neq e''$ implies $\text{LabelOf}_e(e') \neq \text{LabelOf}_e(e'')$) since this is the intended meaning at program analysis time. This, however, would lead to problems when considering soundness: injectiveness is not preserved by arbitrary β-reduction.

If $\text{LabelOf}_e(e') = l$ we often write l as an *annotation* on e'. In other words, we will often use the following syntax[4]

$$\begin{array}{rcl} e & ::= & x \mid \lambda^l x.e \mid e@^l e' \mid \text{fix}^l x.e \mid \text{let}^l\ x = e \text{ in } e' \mid \\ & & \text{if}^l\ e \text{ then } e' \text{ else } e'' \mid \text{True}^l \mid \text{False}^l \mid \\ & & (e, e')^l \mid \text{let}^l\ (x, y) \text{ be } e \text{ in } e' \end{array}$$

We will assume variable names to be distinct, so there is no need for labels on variables[5]. We consider two labelled expressions syntactically equivalent only if their labels are also equivalent.

1.7.1 Flow Properties

We use L to denote finite subsets of $\mathcal{L}$ and $\alpha, \beta, \cdots$ to be variables ranging over sets of labels. We use ℓ to denote sets of labels or label variables:

$$\mathcal{FP} \ni \ell ::= L \mid \alpha \mid \ell \sqcup \ell$$

We call ℓ a *flow property*. Flow properties are an abstraction of a set of values: flow property L is an abstract description of the set of expressions with a label l in L, while α describes an unknown set of abstractions (typically, under some given constraints). We include the least upper bound operator $\sqcup$ in our language of properties. It should become clear below, why this is convenient (we note that that is does *not* make our type based analyses stronger (intuitively because any annotation $\ell \sqcup \ell'$ can be replaced by a fresh variable α and constraints $\ell \leq \alpha$ and $\ell \leq \alpha'$), it does, however, allow a smooth definition of soundness and, later, $\sqcup$ also helps the definition of minimality — this will be crucial in chapter 5.

In chapters 2, 3, 5 and 6, flow properties will be used to annotate type constructors and will therefore often be referred to as *annotations*.

A constraint is a pair of annotations written $\ell \subseteq \ell'$. A *constraint set* C is a set of constraints of the form $\ell \subseteq \alpha$.

The logical rules of figure 1.2 define judgements $C \vdash \ell \subseteq \ell'$.[6] This logic will be common to all the type systems presented in later chapters except chapter 6

A *label substitution* is a map from label variables to flow properties. We use $[\ell/\alpha]$ to denote the substitution mapping α to ℓ while being the identity on all other variables. We use Id to denote the identity substitution. We use the shorthand $S(\kappa)$, $S(C)$ and $S(A)$ when mapping the substitution S

[4] Since expressions that introduce or eliminate values are our prime concern, we often leave out labels from let $x = e$ in e' and fix $x.e$.

[5] We will later need to distinguish different occurrences of variables, but since labels does *not* give a unique index they are not suitable for this purpose, and we will introduce appropriate terminology for this situation.

[6] There is an overlap between the (Id) and (Ax) rules, but we find this presentation more readable.

$$\text{Ax}\ \frac{}{C \vdash L_1 \subseteq L_2}\ \text{if } L_1 \subseteq L_2 \qquad \text{Ax}\ \frac{}{C, \ell \subseteq \alpha \vdash \ell \subseteq \alpha}$$

$$\text{Id}\ \frac{}{C \vdash \ell \subseteq \ell} \qquad \text{Trans}\ \frac{C \vdash \ell_1 \subseteq \ell_2 \quad C \vdash \ell_2 \subseteq \ell_3}{C \vdash \ell_1 \subseteq \ell_3}$$

$$\sqcup\text{-I}\ \frac{}{C \vdash \ell_1 \subseteq \ell_1 \sqcup \ell_2} \qquad \sqcup\ \frac{C \vdash \ell_1 \subset \ell \quad C \vdash \ell_2 \subset \ell}{C \vdash \ell_1 \sqcup \ell_2 \subseteq \ell}$$

$$\text{Assoc}\ \frac{}{C \vdash (\ell_1 \sqcup \ell_2) \sqcup \ell_3 \subseteq \ell_1 \sqcup (\ell_2 \sqcup \ell_3)}$$

$$\text{Comm}\ \frac{}{C \vdash \ell_1 \sqcup \ell_2 \subseteq \ell_2 \sqcup \ell_1}$$

Figure 1.2: Logic of Properties

over κ, C and A resp. (note that even if C is a constraint set $S(C)$ need not be)[7]. The shorthand $C \vdash C'$ means $C \vdash \ell \subseteq \ell'$ for all $\ell \subseteq \ell'$ in C'.

We say that S *solves* C if $\vdash S(C)$ (i.e. for all $\ell \subseteq \ell$ in $S(C)$, we have $\{\} \vdash \ell \subseteq \ell$). Note, that by definition, all constraint sets have solutions.

1.7.2 Flow Functions

Let Destructors(e) be

$$\{l \in \mathcal{L}_e \mid \exists e_1, e_2, e_3.\ \begin{array}{l}\text{ExpOf}_e(l) \supseteq \{\text{if } e_1 \text{ then } e_2 \text{ else } e_3\} \vee \\ \text{ExpOf}_e(l) \supseteq \{\text{let } (x, y) \text{ be } e_1 \text{ in } e_2\} \vee \\ \text{ExpOf}_e(l) \supseteq \{e_1 @ e_2\}\ \}\end{array}$$

and Constructors(e) be

$$\{l \in \mathcal{L}_e \mid \exists x, e_1, e_2.\ \begin{array}{l}\text{ExpOf}_e(l) \supseteq \{\text{True}\} \vee \\ \text{ExpOf}_e(l) \supseteq \{\text{False}\} \vee \\ \text{ExpOf}_e(l) \supseteq \{(e_1, e_2)\} \vee \\ \text{ExpOf}_e(l) \supseteq \{\lambda x.e_1\}\ \}\end{array}$$

Now, a *flow function* of e is a function $\mathcal{F}$: Destructors(e) $\rightarrow \mathcal{FP}$. The intuition is that $\mathcal{F}(l)$ is the set of values that can be used at the destructors labelled l.

[7] We will use postfix notation when applying substitutions $[\ell/\alpha]$ and prefix when applying named substitutions S — this should not be the cause of confusion as the two forms of notation are not used together.

For each of the reductions

$$\begin{array}{rcll} (\lambda^{l'} x.e_1)@^l e_2 & \longrightarrow & e_1[e_2/x] & \text{, or} \\ \text{if}^l\ \text{True}^{l'}\ \text{then}\ e_1\ \text{else}\ e_2 & \longrightarrow & e_1 & \text{, or} \\ \text{if}^l\ \text{False}^{l'}\ \text{then}\ e_1\ \text{else}\ e_2 & \longrightarrow & e_2 & \text{, or} \\ \text{let}^l\ (x,y)\ \text{be}\ (e_1,e_2)^{l'}\ \text{in}\ e_3 & \longrightarrow & e_3[e_1/x][e_2/y] & \end{array}$$

we say that destructor l *consumes* constructor l'.

We say that $\mathcal{F}$ is a *sound* flow of $e \in \text{Exp}$ iff whenever a reduction on some reduction path from $e \longrightarrow \cdots \longrightarrow e' \longrightarrow \cdots$ lets l consume l' then $\vdash \{l'\} \subseteq \mathcal{F}(l)$. If $\mathcal{F}$ is sound for e we write $\mathcal{F} \models e$.

If C is a set of constraints over flow variables, we generalise the notion of soundness such that $C; \mathcal{F} \models e$ iff $S \circ \mathcal{F} \models e$ for all closing substitutions S satisfying C.

Note that a flow function needs to predict all redexes under any reduction strategy. E.g. the flow $\mathcal{F}$ for $(\lambda^{l_1} x.\text{True}^{l_2})@^{l_3}((\lambda^{l_4} y.y)@^{l_5}\text{False}^{l_6})$ needs to map l_5 to $\{l_4\}$ though this would not be reduced under call-by-name.

Our definition of soundness only concerns potential redexes in the analysed term; it does not state anything about redexes arising when applying the term or instantiating free variables. In particular, it allows free variables to be labelled with any set of flow labels, e.g. the empty set. A special consequence of this is that the flow mapping all destructive labels to the empty set is a sound flow for expressions in normal form.

Thus soundness of the flow function is not sufficient to ensure that a flow analysis is reasonable. An important additional constraint will be *principality* that allows the result of flow analysis to be sound in *any* context. This entails that the analysis can be performed *modularly*.

Part I

Monovariant Analysis

Chapter 2

Simple Types

This chapter will introduce the reader to type based flow analysis by defining the simplest imaginable nontrivial flow analysis. The chapter will thus familiarise the reader with the concepts of annotated type system (section 2.1), principality and minimality (section 2.2) and soundness (section 2.3). Section 2.4 presents "CFA via Equality" [Hei95] which is a constraint based analysis of similar precision to our analysis. Section 2.5 discusses implementation and complexity of the presented analyses.

2.1 The Type System

Formulae:

$$\text{Bool}\ \frac{}{\text{Bool}^{\ell} \in \mathcal{K}^s(\text{Bool})}$$

$$\to\ \frac{\kappa \in \mathcal{K}^s(t) \quad \kappa' \in \mathcal{K}^s(t')}{\kappa \to^{\ell} \kappa' \in \mathcal{K}^s(t \to t')} \qquad \times\ \frac{\kappa \in \mathcal{K}^s(t) \quad \kappa' \in \mathcal{K}^s(t')}{\kappa \times^{\ell} \kappa' \in \mathcal{K}^s(t \times t')}$$

Figure 2.1: Simple flow analysis — formulae

Figure 2.1 defines the *formulae* of our flow logic; we will also refer to these as *flow types* or *annotated types*. A set $\mathcal{K}^s(t)$ contains the formulae over a standard type t. We let κ range over members of $\mathcal{K}^s(t)$. The superscript s on sets of formulae is for *simple* and is used to distinguish these formulae from formulae for later analyses.

The *erasure* $|\kappa|$ of an annotated type κ is the standard type obtained by erasing all annotations, formally $\kappa \in \mathcal{K}(t)$ implies $|\kappa| = t$. Erasure

extends its definition to environments A. We define the function ann to map annotated types to their "top" annotation: $ann(\text{Bool}^\ell) = \ell$, $ann(\kappa \to^\ell \kappa') = \ell$ and $ann(\kappa \times^\ell \kappa') = \ell$. An annotation occurs positively resp. negatively in κ iff it annotates a positive resp. negative occurrence of a type constructor in $|\kappa|$.

An *environment* A maps program variables to formulae. A judgement takes the form $C; A \vdash^s e : \kappa$ which should be read "under assumption that the inequalities in C and the assumptions in A are true, it holds that e has property κ".

We will require that if $C; A \vdash^s e : \kappa$ then $|A| \vdash e : |\kappa|$. In other words, flow judgements are annotations of standard type judgements with an additional premise C containing inequalities on annotations only.

Non-logical rules

$$\text{Id}\ \frac{}{C; A, x : \kappa \vdash^s x : \kappa}$$

$$\to\text{-I}\ \frac{C; A, x : \kappa \vdash^s e : \kappa' \quad C \vdash \{l\} \subseteq \ell}{C; A \vdash^s \lambda^l x.e : \kappa \to^\ell \kappa'}$$

$$\to\text{-E}\ \frac{C; A \vdash^s e : \kappa' \to^\ell \kappa \quad C; A \vdash^s e' : \kappa'}{C; A \vdash^s e@^l e' : \kappa}$$

$$\text{Bool-I}\ \frac{C \vdash \{l\} \subseteq \ell}{C; A \vdash^s \text{True}^l : \text{Bool}^\ell} \qquad \frac{C \vdash \{l\} \subseteq \ell}{C; A \vdash^s \text{False}^l : \text{Bool}^\ell}$$

$$\text{Bool-E}\ \frac{C; A \vdash^s e : \text{Bool}^\ell \quad C; A \vdash^s e' : \kappa \quad C; A \vdash^s e'' : \kappa}{C; A \vdash^s \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

$$\times\text{-I}\ \frac{C; A \vdash^s e : \kappa \quad C; A \vdash^s e' : \kappa' \quad C \vdash \{l\} \subseteq \ell}{C; A \vdash^s (e, e')^l : \kappa \times^\ell \kappa'}$$

$$\times\text{-E}\ \frac{C; A \vdash^s e : \kappa \times^\ell \kappa' \quad C; A, x : \kappa, y : \kappa' \vdash^s e' : \kappa''}{C; A \vdash^s \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{fix}\ \frac{C; A, x : \kappa \vdash^s e : \kappa}{C; A \vdash^s \text{fix}\, x.e : \kappa} \qquad \text{let}\ \frac{C; A \vdash^s e : \kappa \quad C; A, x : \kappa \vdash^s e' : \kappa'}{C; A \vdash^s \text{let}\ x = e\ \text{in}\ e' : \kappa'}$$

Figure 2.2: Simple flow analysis — non logical rules

Figure 2.2 presents the non-logical rules for simple flow analysis. The

rules should be straightforward: the only difference to standard simple type rules is that we in all introduction rules make sure that the set of labels on the type of the constructed value contains the annotation of the constructor.

2.1.1 Simple Flow Functions

We have not made the flow information computed by the inference explicit. We can regard the inference tree itself as the result of the analysis: the set of values that an expression e can evaluate to is described by the set of labels on the top constructor of its type. Each judgement $C; A \vdash^s e : \kappa$ in the inference tree approximates the set of values that e can evaluate to by $ann(\kappa)$.

We can compute a flow function from a derivation as follows. If $\mathcal{T}$ is derivation, then for every l we let $\mathcal{F}_{\mathcal{T}}(l)$ be the least annotation such that whenever one of the rules

$$\to\text{-E}\ \frac{C; A \vdash^s e : \kappa' \to^\ell \kappa \quad C; A \vdash^s e' : \kappa'}{C; A \vdash^s e @^l e' : \kappa}$$

$$\times\text{-E}\ \frac{C; A \vdash^s e : \kappa \times^\ell \kappa' \quad C; A, x : \kappa, y : \kappa' \vdash^s e' : \kappa''}{C; A \vdash^s \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{Bool-E}\ \frac{C; A \vdash^s e : \text{Bool}^\ell \quad C; A \vdash^s e' : \kappa \quad C; A \vdash^s e'' : \kappa}{C; A \vdash^s \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

is an inference in $\mathcal{T}$ then $C \vdash \ell \leq \mathcal{F}_{\mathcal{T}}(l)$.

The inference tree will contain more information than the flow function, mapping subexpressions to sets of labels (in particular it allows reasoning about *principality*) so reasoning with inference trees instead of flow functions will often prove rewarding.

2.2 Principality and Minimality

The analysis presented here differs from many constraint and type based analyses by including variables in the language of annotations. Other analyses often aim at finding the *minimal* annotation, but this seems highly problematic, as this will lead to analysing programs under minimal assumptions on free variables and arguments.

For flow analysis this can lead to non-sensical results such as "under assumption that x is annotated with the empty set of labels, the program $(\lambda y.y)@x$ will result in the empty set of labels" which is of course true, but since we will never be able to run the program on any value with the empty set of labels (no such value exists) the result is of little value. In chapter 10 we will see how constant propagation based on flow analysis

can lead to unsound results using a minimal annotation. The problem is that a minimal solution does not capture the flow of input values into the program, and we might be lead to conclude erroneously that variable can only be bound to one constant value, even though it might also bind input values.[1]

What is needed is a *principal* annotation which gives the most general description of the analysed program. In the example above, we could find "under assumption that x is annotated with α, the program $(\lambda y.y)@x$ will result in α".

Amongst the principal typings, we are interested in finding the *minimal* one — intuitively this corresponds to resolving all flow information that does not depend on input or free variables.

2.2.1 Principality

We first note that the inference rules of figure 2.2 are syntax directed. This allows us to directly specify a version of algorithm $\mathcal{W}$ computing a typing [DM82]. We then prove that the typing computed by the algorithm is principal. Algorithm $\mathcal{W}$ maps an environment A and an expression e to a triple of a substitution S, a constraint set C and a flow type κ. The algorithm is defined in figure 2.3.

Unification is defined by the (overloaded) partial function *unify*:

$$\begin{array}{rcll} \mathit{unify}(\alpha,\beta) & = & [\beta/\alpha] & \\ \mathit{unify}(\alpha,L) & = & [L/\alpha] & \\ \mathit{unify}(L,\alpha) & = & [L/\alpha] & \\ \mathit{unify}(L_1,L_2) & = & \mathit{Id} & \text{, if } L_1 = L_2 \\ & = & \mathit{Fail} & \text{, otherwise} \end{array}$$

$$\begin{array}{rcll} \mathit{unify}(\mathrm{Bool}^{\ell_1},\mathrm{Bool}^{\ell_2}) & = & \mathit{unify}(\ell_1,\ell_2) & \\ \mathit{unify}(\kappa_1 \to^{\ell_1} \kappa_1', \kappa_2 \to^{\ell_2} \kappa_2') & = & \text{let} & S_1 = \mathit{unify}(\kappa_1,\kappa_2) \\ & & & S_2 = \mathit{unify}(S_1\kappa_1', S_1\kappa_2') \\ & & & S_3 = \mathit{unify}(S_2(S_1\ell_1), S_2(S_1\ell_2)) \\ & & \text{in} & S_3 \circ S_2 \circ S_1 \\ \mathit{unify}(\kappa_1 \times^{\ell_1} \kappa_1', \kappa_2 \times^{\ell_2} \kappa_2') & = & \text{let} & S_1 = \mathit{unify}(\kappa_1,\kappa_2) \\ & & & S_2 = \mathit{unify}(S_1\kappa_1', S_1\kappa_2') \\ & & & S_3 = \mathit{unify}(S_2(S_1\ell_1), S_2(S_1\ell_2)) \\ & & \text{in} & S_3 \circ S_2 \circ S_1 \end{array}$$

Proposition 2.1 *The definition of unify implements* most general unification*:*

[1] Previous flow analyses have been global and focused on tracing the flow of higher-order values: under the assumption that all input is first-order, minimal solutions *does* make sense.

$\mathcal{W}(A, e) =$ case e of

x :	$(Id, \{\,\}, A(x))$
$\lambda^l x : t.e'$:	let $\kappa \in \mathcal{K}^s(t)$ be a flow type with fresh variable annotations let α be a fresh variable let $(S, C, \kappa') = \mathcal{W}((A, x : \kappa), e')$ in $(S, C \cup \{\{l\} \subseteq \alpha\}, S(\kappa) \to^\alpha \kappa')$
$e_1 @^l e_2$:	let $(S, C, \kappa'' \to^\ell \kappa) = \mathcal{W}(A, e_1)$ let $(S', C', \kappa') = \mathcal{W}(SA, e_2)$ let $S'' = \mathit{unify}(\kappa', S'\kappa'')$ in $(S'' \circ S' \circ S, S''(S'C \cup C'), S''(S'\kappa))$
$\mathrm{True}^l(\mathrm{False}^l)$:	let α be a fresh variable in $(Id, \{\{l\} \subseteq \alpha\}, \mathrm{Bool}^\alpha)$
if^l e_1 then e_2 else e_3 :	let $(S_1, C_1, \mathrm{Bool}^\ell) = \mathcal{W}(A, e_1)$ let $(S_2, C_2, \kappa_2) = \mathcal{W}(S_1 A, e_2)$ let $(S_3, C_3, \kappa_3) = \mathcal{W}(S_2(S_1 A), e_3)$ let $S_4 = \mathit{unify}(S_3\kappa_2, \kappa_3)$ in $(S_4 \circ S_3 \circ S_2 \circ S_1, S_4(S_3(S_2 C_1 \cup C_2) \cup C_3), S_4\kappa_3)$
$(e_1, e_2)^l$:	let α be a fresh variable let $(S_1, C_1, \kappa_1) = \mathcal{W}(A, e_1)$ let $(S_2, C_2, \kappa_2) = \mathcal{W}(S_1 A, e_2)$ in $(S_2 \circ S_1, S_2 C_1 \cup C_2 \cup \{\{l\} \subseteq \alpha\}, (S_2\kappa_1) \times^\alpha \kappa_2)$
let^l (x, y) be e_1 in e_2 :	let $(S_1, C_1, \kappa_x \times^\ell \kappa_y) = \mathcal{W}(A, e_1)$ let $(S_2, C_2, \kappa) = \mathcal{W}((S_1 A, x : \kappa_x, y : \kappa_y), e_2)$ in $(S_2 \circ S_1, S_2 C_1 \cup C_2, \kappa)$
fix $x : t.e'$:	let $\kappa \in \mathcal{K}^s(t)$ be a flow type with fresh variable annotations let $(S, C, \kappa') = \mathcal{W}((A, x : \kappa), e')$ let $S' = \mathit{unify}(S\kappa, \kappa')$ in $(S' \circ S, S'C, S'\kappa')$
let $x = e_1$ in e_2 :	let $(S_1, C_1, \kappa) = \mathcal{W}(A, e_1)$ let $(S_2, C_2, \kappa') = \mathcal{W}(S_1(A, x : \kappa), e_2)$ in $(S_2 \circ S_1, S_2 C_1 \cup C_2, \kappa')$

Figure 2.3: Algorithm $\mathcal{W}$ for simple types

1. *If there exists S such that $S\kappa = S\kappa'$ then $unify(\kappa, \kappa')$ does not fail.*
2. *If $S = unify(\kappa, \kappa')$ then $S\kappa = S\kappa'$ and for any S' such that $S'\kappa = S'\kappa'$ there exists S'' such that $S' = S'' \circ S$.*

We observe that $\mathcal{W}(A, e)$ can fail for some A (since unification can fail). This will of course happen if no t exists such that $|A| \vdash e : t$ (that is e is not well typed under the standard type component of A). But even if A is a decoration of a valid standard environment, the algorithm can fail. E.g. $\mathcal{W}(f : \mathrm{Bool}^{\{l_3\}} \rightarrow^{\{l_4\}} \mathrm{Bool}^{\{l_5\}}, f@^{l_1}\mathrm{True}^{l_2})$ will fail since $\{l_2\}$ and $\{l_3\}$ are not unifiable. We introduce a sufficient criterion on environments to avoid that $\mathcal{W}$ fails:

Definition 2.2 *We call A* proper *if for all $x \in \mathrm{Dom}(A)$, all annotations in $A(x)$ are label variables.*

We could have chosen a less restrictive definition of proper environments — when we introduce subtyping in the next chapter, we will show that requiring the negative occurrences in κ_i to be label variables suffices. The call

$$\mathcal{W}(x : \mathrm{Bool}^{\{l_1\}}, y : \mathrm{Bool}^{\{l_2\}}, \text{if } \mathrm{True}^{l_3} \text{ then } x \text{ else } y)$$

shows this to be insufficient for simple flow analysis since $\{l_1\}$ and $\{l_2\}$ are not unifiable.

Lemma 2.3 *If A is proper then $\mathcal{W}(A, e)$ terminates without failure.*

Proof We prove by easy induction that if A is proper then $\mathcal{W}(A, e)$ returns (S, C, κ) where κ is annotated with flow variables only and the range of S is only flow variables.

□

We are now able to define the *instance* relation between typings:

Definition 2.4 *A flow judgement $C'; A' \vdash^s e' : \kappa'$ is an (weak)* instance *of $C; A \vdash^s e : \kappa$ if $e = e'$ and there exists a substitution S such that*

1. *$C' \vdash^s S(C)$,*
2. *$A'(x) = S(A(x))$ for all $x \in \mathrm{Dom}(A)$ and*
3. *$S(\kappa) = \kappa'$*

If $C; A \vdash^s e : \kappa$ is also an instance of $C'; A' \vdash^s e' : \kappa'$ then the two judgements are equivalent.

Proposition 2.5 *Derivable judgements are closed under the instance relation. That is, if $C; A \vdash^s e : \kappa$ is derivable then so is any instance of it.*

Proof By induction over the derivation of $C; A \vdash^s e : \kappa$. □

Lemma 2.6 *If $C; A \vdash^s e : \kappa$ then also $C, C'; A \vdash^s e : \kappa$ for any constraint set C'.*

Proof Follows from proposition 2.5. □

Theorem 2.7 *If $\mathcal{W}(A, e) = (S, C, \kappa)$ then $C; S(A) \vdash^s e : \kappa$ and for any C', κ', S' where $C'; S'(A) \vdash^s e : \kappa'$ we have that $C'; S'(A) \vdash^s e : \kappa'$ is an instance of $C; S(A) \vdash^s e : \kappa$.*

Proof The theorem follows by induction over the structure of e. We show a few cases, the rest are similar:

"$e = x$": We find $\mathcal{W}(A, e) = (Id, \{\}, A(x))$. If we assume $A = A', x : \kappa$, we have

$$\frac{}{; A', x : \kappa \vdash^s x : \kappa}$$

Furthermore, let C', κ', S' be given such that $C'; S'(A) \vdash^s x : \kappa'$. Then $S'(A) = S'(A'), x : S'(\kappa)$ and $\kappa' = S'(\kappa)$. We have:

1. $C' \vdash^s S'\{\}$,
2. $S'(A(x)) = S'(A(x))$ for all $x \in \mathrm{Dom}(A)$, and
3. $S'(\kappa) = \kappa'$

and thus $C'; S'(A) \vdash^s x : \kappa'$ is an instance of $; A \vdash^s x : \kappa$.

"$e = \lambda^l x : t.e'$": Let $\kappa \in \mathcal{K}^s(t)$ be a flow type with fresh variable annotations, α be a fresh variable and let $(S, C, \kappa') = \mathcal{W}((A, x : \kappa), e')$. By induction, $C; S(A, x : \kappa) \vdash^s e' : \kappa'$ and by lemma 2.6 also

$$C, \{l\} \subseteq \alpha; S(A, x : \kappa) \vdash^s e' : \kappa'$$

By the ($\rightarrow$-I) rule we find

$$C, \{l\} \subseteq \alpha; S(A) \vdash^s \lambda^l x : t.e' : S(\kappa) \rightarrow^\alpha \kappa' \tag{2.1}$$

Let $C', \kappa_1, \kappa_2, \ell, S'$ be given such that

$$C'; S'(A) \vdash^s \lambda^l x : t.e' : \kappa_1 \rightarrow^\ell \kappa_2 \tag{2.2}$$

The last rule applied must be the ($\rightarrow$-I) rule from assumptions

$$C'; S'(A), x : \kappa_1 \vdash^s e' : \kappa_2 \quad \text{and} \quad C' \vdash^s \{l\} \subseteq \ell$$

Without loss of generality[2], we can assume that S' is the identity on κ_1, which allows us to use the induction hypothesis to show that $C'; S'(A, x : \kappa_1) \vdash^s e' : \kappa_2$ is an instance of $C, \{l\} \subseteq \alpha; S(A, x : \kappa) \vdash^s e' : \kappa'$ that is there exists S'' such that:

1. $C' \vdash^s S''(C)$
2. $S'((A, x : \kappa_1)(y)) = S''(S(A, x : \kappa)(y))$ for all $y \in \mathrm{Dom}(A, x : \kappa)$, and
3. $S''(\kappa') = \kappa_2$

To show that (2.2) is an instance of (2.1) we let $S''' = S'' \circ [\ell/\alpha]$ and have:

1. $C' \vdash^s S'''(C)$
2. $S'(A(y)) = S'''(S(A(y)))$ for all $y \in \mathrm{Dom}(A)$, and
3. $S'''(S(\kappa) \to^\alpha \kappa') = \kappa_1 \to^\ell \kappa_2$

"$e = e_1@^l e_2$": Let $(S, C, \kappa'' \to^\ell \kappa) = \mathcal{W}(A, e_1)$ and $(S', C', \kappa') = \mathcal{W}(SA, e_2)$. By induction, we find

$$C; S(A) \vdash^s e_1 : \kappa'' \to^\ell \kappa \tag{2.3}$$

(note that e_1 must have a function type due to standard well-typedness) and

$$C'; S'(S(A)) \vdash^s e_2 : \kappa' \tag{2.4}$$

Let $S'' = \mathit{unify}(\kappa', S'\kappa'')$. Since

$$(S'' \circ S')C; (S'' \circ S' \circ S)(A) \vdash^s e_1 : (S'' \circ S')(\kappa'' \to^\ell \kappa) \tag{2.5}$$

is an instance of (2.3) we have by proposition 2.5 that it is also derivable and then by lemma 2.6

$$S''(S'C \cup C'); (S'' \circ S' \circ S)(A) \vdash^s e_1 : (S'' \circ S')(\kappa'' \to^\ell \kappa) \tag{2.6}$$

is also derivable.

Similarly

$$S''(S'C \cup C'); (S'' \circ S' \circ S)(C \cup C')(A) \vdash^s e_2 : S''(\kappa') \tag{2.7}$$

[2]Formally, we show that there exists a renaming S_r s.t.

$$C'; S'(A) \vdash^s \lambda^l x : t.e' : \kappa_1 \to^\ell \kappa_2$$

and

$$S_r C'; S'(A) \vdash^s \lambda^l x : t.e' : S_r(\kappa_1 \to^\ell \kappa_2)$$

are equivalent. The proof then proceeds with the second judgement.

By proposition 2.1 we find that $S''\kappa' = S''(S'\kappa'')$ and hence by the ($\rightarrow$-E) rule:

$$S''(S'C \cup C'); (S'' \circ S' \circ S)(A) \vdash^s e_1@^l e_2 : S''(S'\kappa) \tag{2.8}$$

Let C_1, κ_1, S_1 be given such that

$$C_1; S_1(A) \vdash^s e_1@^l e_2 : \kappa_1$$

The last rule applied must be ($\rightarrow$-E) and hence there exist κ_2 and ℓ' such that

$$C_1; S_1(A) \vdash^s e_1 : \kappa_2 \rightarrow^{\ell'} \kappa_1$$

and

$$C_1; S_1(A) \vdash^s e_2 : \kappa_2$$

By induction hypothesis we find S_3 s.t.

1. $C_1 \vdash^s S_3(C)$,
2. $S_1(A(x)) = S_3(S(A(x)))$ for all $x \in \mathrm{Dom}(A)$, and
3. $S_3(\kappa'' \rightarrow^{\ell} \kappa) = \kappa_2 \rightarrow^{\ell'} \kappa_1$

and similarly by induction hypothesis S_4 s.t.

1. $C_1 \vdash^s S_4(C')$,
2. $S_1(A(x)) = S_4(S'(S(A(x))))$ for all $x \in \mathrm{Dom}(A)$, and
3. $S_4(\kappa') = \kappa_2$

Let $\{\alpha_1, \cdots, \alpha_n\}$ be the variables in $S(A)$. From the two point 2.'s we see that for all α_i we have $(S_4 \circ S')(\alpha_i) = S_3(\alpha_i)$. Let the variables in $\kappa'' \rightarrow^{\ell} \kappa$ be $\{\beta_1, \cdots, \beta_m\}$ and the variables of κ' be $\{\gamma_1, \cdots, \gamma_k\}$. We can assume that

1. $\{\beta_1, \cdots, \beta_m\} \cap \{\gamma_1, \cdots, \gamma_k\} \subseteq \{\alpha_1, \cdots, \alpha_n\}$,
2. $(\{\beta_1, \cdots, \beta_m\} \setminus \{\alpha_1, \cdots, \alpha_n\}) \cap \mathrm{Dom}(S_4 \circ S') = \emptyset$,
3. $(\{\gamma_1, \cdots, \gamma_k\} \setminus \{\alpha_1, \cdots, \alpha_n\}) \cap \mathrm{Dom}(S_3) = \emptyset$, and
4. $\mathrm{Ran}(S_4) \cap \mathrm{Dom}(S_3) \subseteq \{\alpha_1, \cdots, \alpha_n\}$

We now find

$$\begin{array}{rcll} (S_3 \circ S_4)(S'\kappa'') & = & S_3((S_4 \circ S')\kappa'') & \\ & = & (S_4 \circ S')(S_3\kappa'') & (\dagger) \\ & = & (S_4 \circ S')\kappa_2 & \\ & = & (S_4 \circ S')(S_4\kappa') & \\ & = & S_3(S_4\kappa') & (\ddagger) \\ & = & (S_3 \circ S_4)\kappa' & \end{array}$$

where (†) follows from point 2. and (‡) follows from point 4. and the fact mentioned above that $(S_4 \circ S')(\alpha_i) = S_3(\alpha_i)$.

It follows from proposition 2.1 that there exists S_5 such that:

$$S_3 \circ S_4 = S_5 \circ S''$$

It now follows that

1. $C_1 \vdash^s S_5(S''(S'C \cup C'))$,
2. $S_1(A(x)) = S_5((S'' \circ S' \circ S)(A(x)))$ for all $x \in \mathrm{Dom}(A)$, and
3. $S_5(S''(S'\kappa)) = \kappa_1$

where the first two points follow by simple calculations and point 3. follows from

$$\begin{array}{rcl} S_5(S''(S'\kappa)) & = & (S_5 \circ S'')(S'\kappa) \\ & = & (S_3 \circ S_4)(S'\kappa) \\ & = & S_3((S_4 \circ S')\kappa) \\ & = & \kappa_1 \end{array}$$

The last step is true because of point 2. and $(S_4 \circ S')(\alpha_i) = S_3(\alpha_i)$.

□

This theorem along with lemma 2.3 shows that every expression has principal types under proper assumptions.

2.2.2 Minimality

Principal typings are *not* unique: $\mathcal{W}([\,], \mathrm{if}^{l_4}\ \mathrm{True}^{l_1}\ \mathrm{then}\ \mathrm{False}^{l_2}\ \mathrm{else}\ \mathrm{True}^{l_3})$ will return $(Id, \{\{l_1\} \subseteq \alpha, \{l_2\} \subseteq \beta, \{l_3\} \subseteq \beta\}, \mathrm{Bool}^{\beta})$ corresponding to the typing

$$\{l_1\} \subseteq \alpha, \{l_2\} \subseteq \beta, \{l_3\} \subseteq \beta; \vdash \mathrm{if}^{l_4}\ \mathrm{True}^{l_1}\ \mathrm{then}\ \mathrm{False}^{l_2}\ \mathrm{else}\ \mathrm{True}^{l_3} : \mathrm{Bool}^{\beta}$$

The first constraint signifies that *any* set of labels can be used by the 'if' as long as l_1 is amongst them. We would prefer a principal typing

$$\{l_2\} \subseteq \beta, \{l_3\} \subseteq \beta; \vdash \mathrm{if}^{l_4}\ \mathrm{True}^{l_1}\ \mathrm{then}\ \mathrm{False}^{l_2}\ \mathrm{else}\ \mathrm{True}^{l_3} : \mathrm{Bool}^{\beta}$$

where we have instantiated α to $\{l_1\}$, since this is the best (minimal) information we can obtain without violating principality. If we compute the flow function for the two derivations, the first corresponds to $[l_4 \mapsto \alpha]$ and the second to $[l_4 \mapsto \{l_1\}]$ where the first is more general, but more general in an "unnecessary" way: no evaluation of the expression can result in the 'if' using any other value than True^{l_1}.

It is easy to see that if $(S, C, \kappa) = \mathcal{W}(A, e)$ and A is proper then all constraints in C have the form $\{l\} \subseteq \alpha$. Let α be a label variable occurring in C. Let $\{l_1\} \subseteq \alpha, \cdots, \{l_n\} \subseteq \alpha$ be all constraints in C containing α. If α

occurs neither in A nor κ, let C' be the result of removing these constraints from C. If α occurs in A or κ, let C' be the result of replacing the constraints with $\bigcup_i\{l_i\} \subseteq \alpha$. In either case $C'; A \vdash e : \kappa$ and $C; A \vdash e : \kappa$ are equivalent (instances of each other).

By repeating this procedure, we obtain:

Theorem 2.8 *For every flow judgement $C; A \vdash e : \kappa$ there exists an equivalent judgement $C'; A \vdash e : \kappa$ where C' contains only flow variables α occurring free in A or κ and for each α the constraint set C' contains only one constraint $L \subseteq \alpha$.*

If we keep track of labels being used in the program (e.g. at the 'if' statement note which labels annotate the boolean being consumed) during computation of $\mathcal{W}(A, e)$ then applying substitutions $S = [\bigcup_i\{l_i\}/\alpha]$ (in the discussion above) to the inference tree (alternatively, the flow function) will yield *minimal*, principal flow information.

The flow function computed this way will be polymorphic, but it has the special property that it will map all destructors to either a constant or to a variable occurring in the environment or in the type of the whole expression. Thus instantiating type and environment will allow flow information to be read directly from the polymorphic flow function.

By searching for a minimal type only amongst the principal types, we differ significantly from constraint based flow analyses by Palsberg [Pal94] and Heintze [Hei95]. Since their analyses do not contain variables, they do not have the principal type property. This prevents their analyses from being modular. They then set out to find the minimal typing which is indeed small but also quite un-informative since it will be the typing that given the smallest assumptions on free variables and input computes the smallest result. Our approach requires the typing to be applicable in *all* contexts and only then chooses the minimal typing amongst these modular descriptions.

2.3 Soundness

This section will prove soundness of the analysis. We will prove the more general result that flow information is preserved under reduction.

First we prove a variant of the substitution lemma: not only is the final judgement preserved by substitution, but also the flow computed by the derivations.

Lemma 2.9 (Substitution lemma) *If* $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C; A, x : \kappa' \vdash^s e : \kappa}$ *and* $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C; A \vdash^s e' : \kappa'}$ *then there exists $\mathcal{T}_\ni$ such that*

1. $\mathcal{T}_3 = \dfrac{\mathcal{T}_3'}{C; A \vdash^s e[e'/x] : \kappa}$ *and*

2. *For all* $l \in \text{Destructors}(e[e'/x])$ *we have*

$$C \vdash \mathcal{F}_{\mathcal{T}_3}(l) = \mathcal{F}_{\mathcal{T}_1}(l) \sqcup \mathcal{F}_{\mathcal{T}'_2}(l)$$

Proof The lemma follows by simple induction on the structure of $\mathcal{T}_1$. We give two illustrative cases:

(Id): Assume

$$\mathcal{T}_1 = \frac{}{C; A, x : \kappa' \vdash^s x : \kappa'}$$

and

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{C; A \vdash^s e' : \kappa'}$$

1. Let $\mathcal{T}_3 = \mathcal{T}_2$
2. Let $l \in \text{Destructors}(x[e'/x])$ be given. Since expression x has no constructors, it follows that $\mathcal{F}_{\mathcal{T}_1}(l) = \emptyset$ and clearly $\mathcal{F}_{\mathcal{T}_2}(l) = \mathcal{F}_{\mathcal{T}_3}(l)$.

($\rightarrow$-E): Assume

$$\mathcal{T}_1 = \frac{\dfrac{\mathcal{T}_1'}{C; A, x : \kappa' \vdash^s e : \kappa'' \rightarrow^\ell \kappa} \qquad \dfrac{\mathcal{T}_1''}{C; A, x : \kappa' \vdash^s e'' : \kappa''}}{C; A, x : \kappa' \vdash^s e @^{l_@} e'' : \kappa''}$$

and

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{C; A \vdash^s e' : \kappa'}$$

then by induction there exists

$$\mathcal{T}_3' = \frac{\mathcal{T}_3''}{C; A \vdash^s e[e'/x] : \kappa'' \rightarrow^\ell \kappa} \quad \text{and} \quad \mathcal{T}_4' = \frac{\mathcal{T}_4''}{C; A \vdash^s e''[e'/x] : \kappa''}$$

Now

1. Let

$$\mathcal{T}_3 = \frac{\dfrac{\mathcal{T}_3''}{C; A \vdash^s e[e'/x] : \kappa'' \rightarrow^\ell \kappa} \qquad \dfrac{\mathcal{T}_4''}{C; A \vdash^s e''[e'/x] : \kappa''}}{C; A \vdash^s (e @^{l_@} e'')[e'/x] : \kappa}$$

2. Let $l \in \text{Destructors}((e_1 @^{l_@} e_2)[e'/x])$ be given. If $l \neq l_@$ we are done by straightforward induction. Otherwise

$$\begin{array}{l} C \vdash \mathcal{F}_{\mathcal{T}_3}(l_@) = \mathcal{F}_{\mathcal{T}_3''}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_4''}(l_@) \sqcup \ell \\ C \vdash \mathcal{F}_{\mathcal{T}_3''}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_4''}(l_@) \sqcup \ell \\ \quad = \mathcal{F}_{\mathcal{T}_1'}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_2}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_1''}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_2}(l_@) \sqcup \ell \\ C \vdash \mathcal{F}_{\mathcal{T}_1'}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_2}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_1''}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_2}(l_@) \sqcup \ell \\ \quad = \mathcal{F}_{\mathcal{T}_1}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_2}(l_@) \end{array}$$

where the first step is the definition, the second step is the induction hypothesis and the last step follows from

$$C \vdash \mathcal{F}_{\mathcal{T}_1}(l_@) = \mathcal{F}_{\mathcal{T}_1'}(l_@) \sqcup \mathcal{F}_{\mathcal{T}_1''}(l_@) \sqcup \ell$$

□

We then prove the main subject reduction theorem which is also extended with preservation of types of subexpressions. We call this property *strong* subject reduction.

Theorem 2.10 (Subject Reduction) *If* $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C; A \vdash^s e_1 : \kappa}$ *and* $e_1 \longrightarrow e_2$ *then there exists* $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C; A \vdash^s e_2 : \kappa}$ *and*
2. *For all* $l \in \text{Destructors}(e_2)$ *we have*

$$C \vdash \mathcal{F}_{\mathcal{T}_2}(l) \subseteq \mathcal{F}_{\mathcal{T}_1}(l)$$

and if $l \in \text{Destructors}(e_1)$ *consumes* $l' \in \text{Constructors}(e_1)$ *then*

$$C \vdash \{l'\} \subseteq \mathcal{F}_{\mathcal{T}_1}(l)$$

Proof We show a few cases for illustration:

(β) $(\lambda^l x.e)@^{l'} e' \longrightarrow e[e'/x]$. We have

$$\mathcal{T}_1 = \frac{\mathcal{T}_1'}{C; A \vdash^s (\lambda^l x.e)@^{l'} e' : \kappa}$$

There must exist $\mathcal{T}_1'', \mathcal{T}_1''', \kappa', \alpha$ such that the derivation looks as follows:

$$\mathcal{T}_1 = \dfrac{\dfrac{\dfrac{\mathcal{T}_1''}{C; A, x : \kappa' \vdash^s e : \kappa} \quad C \vdash \{l\} \subseteq \alpha}{C; A \vdash^s \lambda^l x.e : \kappa' \to^\alpha \kappa} \quad \dfrac{\mathcal{T}_1'''}{C; A \vdash^s e' : \kappa'}}{C; A \vdash^s (\lambda^l x.e)@^{l'} e' : \kappa}$$

By lemma 2.9 there exists $\mathcal{T}_2$ such that

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{C; A \vdash^s e[e'/x] : \kappa}$$

and for all $l'' \in \text{Destructors}(e[e'/x])$

$$C \vdash \mathcal{F}_{\mathcal{T}_2}(l'') = \mathcal{F}_{\mathcal{T}_1''}(l'') \sqcup \mathcal{F}_{\mathcal{T}_1'''}(l'')$$

Since

$$\begin{array}{rcll} C \vdash \mathcal{F}_{\mathcal{T}_1}(l'') & = & \mathcal{F}_{\mathcal{T}_1''}(l'') \sqcup \mathcal{F}_{\mathcal{T}_1''}(l'') \sqcup \alpha & \text{if } l'' = l' \\ & = & \mathcal{F}_{\mathcal{T}_1''}(l'') \sqcup \mathcal{F}_{\mathcal{T}_1''}(l'') & \text{otherwise} \end{array}$$

and $C \vdash \{l\} \subseteq \alpha$, point *2.* follows.

(δ-if) if^l True then e else $e' \longrightarrow e$. We have

$$\mathcal{T}_1 = \frac{\mathcal{T}_1'}{C; A \vdash^s \text{if}^l \text{ True then } e \text{ else } e' : \kappa}$$

which must be by the (Bool-I) rule, thus:

$$\mathcal{T}_1 = \frac{\dfrac{C \vdash \{l\} \subseteq \ell}{C; A \vdash^s \text{True}^l : \text{Bool}^\ell} \qquad \dfrac{\mathcal{T}_1''}{C; A \vdash^s e : \kappa} \qquad \dfrac{\mathcal{T}_1'''}{C; A \vdash^s e' : \kappa}}{C; A \vdash^s \text{if}^l \text{ True then } e \text{ else } e' : \kappa}$$

Hence $\mathcal{T}_1''$ is the derivation proving *1.* Point *2.* follows directly.

(Context) $C[e] \longrightarrow C[e']$ where $e \longrightarrow e'$. Follows by induction on the context C.

□

The following corollary follows as an immediate consequence of theorem 2.10:

Corollary 2.11 (Soundness for Simple Flow Analysis) *Let $\mathcal{T}$ be any derivation for e and let $C; A \vdash^s e : \kappa$ be its conclusion. Then $C; \mathcal{F}_{\mathcal{T}} \models e$.*

2.4 Constraint Based Analysis

Heintze defines a constraint based flow analysis for untyped lambda calculus which he calls "CFA via Equality" [Hei95][3]. We will briefly review his analysis here. Our presentation is adapted to our richer language and we will use a notation based on a mixture of Palsberg and O'Keefe [Pal94, PO95].

[3]Heintze defines four different systems and shows equivalences between those and four type based analyses. We will return to the analysis he calls "standard CFA" in the next chapter.

To every subexpression e' occurrence of the analysed program e, we associate a variable which we name $[\![e']\!]$. Let Y_e denote this set of variables and let X_e denote the set of program variables occurring in e (which we by well-namedness assume are distinct).

The constraint generation is shown in figure 2.4. It generates constraints over $X_e \cup Y_e$. (In this analysis, there is no need for the distinction between x and $[\![x]\!]$. In other words, we could generate constraints over Y_e only but we use this notation for consistency with the later constraint based analysis of section 3.4). If $V_1, V_2, V_3, V_4 \in X_e \cup Y_e$ then the constraints have one of the forms $V_1 \subseteq V_2$, $V_1 = V_2$ or $V_1 \subseteq V_2 \Longrightarrow V_3 = V_4$ (in the next chapter we generalise this form to $V_1 \subseteq V_2 \Longrightarrow V_3 \subseteq V_4$). The last form of constraint is called *conditional* and are a kind of closing rules on the generated set of constraints: if $V_1 \subseteq V_2$ is provable then $V_3 = V_4$ should be added to the constraint set.

Generating constraints for e.

For every occurrence in e of:	generate:
x	$x = [\![x]\!]$
$\lambda^l x.e_1$	$\{l\} \subseteq [\![\lambda^l x.e_1]\!]$
$e_1 @ e_2$	for every $\lambda^l x.e_3$ in e:
	$\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_2]\!] = x$
	$\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_3]\!] = [\![e_1 @ e_2]\!]$
True^l	$\{l\} \subseteq [\![\text{True}^l]\!]$
False^l	$\{l\} \subseteq [\![\text{False}^l]\!]$
if e_1 then e_2 else e_3	$[\![e_2]\!] = [\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]$
	$[\![e_3]\!] = [\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]$
$(e_1, e_2)^l$	$\{l\} \subseteq [\![(e_1, e_2)^l]\!]$
let (x, y) be e_1 in e_2	$[\![e_2]\!] = [\![\text{let } (x, y) \text{ be } e_1 \text{ in } e_2]\!]$
	for every $(e_3, e_4)^l$ in e:
	$\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_3]\!] = x$
	$\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_4]\!] = y$
fix $x.e_1$	$[\![e_1]\!] = x$
	$[\![e_1]\!] = [\![\text{fix } x.e_1]\!]$
let $x = e_1$ in e_2	$[\![e_1]\!] = x$
	$[\![e_2]\!] = [\![\text{let } x = e_1 \text{ in } e_2]\!]$

Figure 2.4: Constraint generation a la Heintze

Heintze notes that this analysis is equivalent to the flow analysis of Bondorf and Jørgensen [BJ93]. This requires some insight as the right-hand sides "for every..." in Bondorf and Jørgensen's analysis are represented as explicit constraints (called $\prec_n^*$ and $\prec_C^*$) which are then solved at once in-

stead of looking up all lambda- resp. pair-expressions in the program every time. Thus Bondorf and Jørgensen's representation is harder to read but more directly implementable.

A solution to the constraint set is a substitution mapping the above variables in $X_e \cup Y_e$ to sets of labels such that the constraints are true.

There is a major difference between our analysis and the analyses of Bondorf and Jørgensen and Heintze: we are dealing with a typed language, whereas they were doing analysis of untyped languages: Scheme resp. untyped lambda-calculus. Another difference between our analysis and the analysis presented by Heintze is our introduction of flow variables ranging over sets of labels.

Since our language is typed, the equivalence between simple flow analysis and CFA via Equality only follows indirectly from Heintze's results. This is the subject of the following subsection.

2.4.1 Equivalence between Simple Flow Analysis and CFA via Equality

In addition to the system presented above, Heintze presents a system he calls "CFA via Equality without recursion". This system produces the same constraints as CFA via Equality. A substitution S is a solution to the constraint set only if $\succ$ forms a non-reflexive ordering:

> If $\lambda^l x.e'$ occurs in e and x occurs free in e' then $\forall l' \in S[\![x]\!] : l \succ l'$

E.g. consider $(\lambda y.y@y)@(\lambda^l x.x@x)$ where $S[\![x]\!] = Sx = \{l\}$ and hence $l \succ l$ and thus $\succ$ is reflexive.

Heintze proves that "CFA via Equality without recursion" is at least as powerful as a system corresponding directly to our simple type based flow analysis. His theorem can be reformulated as:

> If $C; A \vdash^s e' : \kappa$ is a ground judgement in the inference tree for e then there exists a solution S to the constraint set such that $ann(\kappa) \subseteq S[\![e']\!]$.

It follows as an immediate consequence that the same theorem is true for "CFA via Equality".

Heintze also proves that the other direction holds for "CFA via Equality without recursion". This can be reformulated as:

> If S is a solution to the constraint set then there exists a ground inference tree $\mathcal{T}$ such that $S[\![e]\!] = ann(\kappa)$ for every e for which the judgement $C; A \vdash^s e : \kappa$ is in $\mathcal{T}$.

Now add type variables k and recursive types $\mu k.\kappa$ to the language of annotated types. Define an equality on types by:

$$\mu k.\kappa = \kappa[\mu k.\kappa/k]$$

and extend this in the obvious way to become a congruence on types. Annotated types in the same equivalence class are identified and free variables are disallowed.

For "CFA via Equality" and the system obtained by adding recursive types to our simple typed flow analysis (call this $\vdash^{\mu}$) the same result as above holds:

> If S is a solution to the constraint set then there exists a ground inference tree $\mathcal{T}$ such that $S[\![e]\!] = ann(\kappa)$ for every e for which the judgement $C; A \vdash^{\mu} e : \kappa$ is in $\mathcal{T}$.

At this point we realise that no (well typed) term exists such that $C; A \vdash^{\mu} e : \kappa$ but not $C; A \vdash^{s} e : \kappa$. Thus $C; A \vdash^{\mu} e : \kappa$ implies $C; A \vdash^{s} e : \kappa$ and we have established the equivalence between the analysis defined by the constraint generation system of figure 2.4 (without the $\succ$-constraint) and our simple flow analysis.

Thus for typed terms, constraint based analysis finds a solution equivalent to the minimal ground derivation in our system.

2.5 Algorithm

Bondorf and Jørgensen [BJ93] show that their simple closure analysis is computable by an algorithm running in time $\mathcal{O}(n\alpha(n,n))$ where $\alpha(n,n) < 5$ for any value of n smaller than the number of atoms in the universe. (To be exact, their complexity contains a few more figures like the maximal arity of functions and constructors, which we can ignore).

The complexity argument is based on an algorithm developed for binding time analysis by Henglein [Hen91]. The factor $\alpha(n,n)$ stems from unification implemented using union/find (α is an inverse of Ackermann's function).

Chapter 3

Subtyping

In this chapter we add subtyping to the system of chapter 2. This allows greater precision of the analysis. The resulting analysis is equivalent in strength to the closure analysis of Sestoft [Ses88, Ses91] (section 3.3) and the constraint based analysis of Palsberg [Pal94] (section 3.4). Palsberg proved the constraint based analysis equal to Sestoft's, and Palsberg and O'Keefe [PO95] and Heintze [Hei95] independently proved the equivalence between constraint based and sub type based analysis. These proofs are sketched in section 3.5.

In contrast to the analyses of Sestoft and Palsberg, our flow analysis can analyse parts of a program independently without loss of precision since it enjoys the principal typing property.

3.1 Subtyping

Consider the following expression

$$
\begin{array}{l}
\text{let } x = \text{True}^{l_1} \\
\text{in } \text{ let } y = \text{False}^{l_2} \\
\qquad \text{in } \text{ let } f = \lambda z.z \\
\qquad\qquad \text{in } \text{ if } x \\
\qquad\qquad\qquad \text{then } f@x \\
\qquad\qquad\qquad \text{else } f@y
\end{array}
$$

Variable f can be applied to both x and y and thus using the simple analysis of chapter 2, the flow information associated with these two variables has to be identical. Hence the analysis tells us that the condition of the 'if' can evaluate to either True^{l_1} or False^{l_2}.

Subtyping will allow us to associate exact information to x and y and then only subtype this to $\{l_1, l_2\}$ when x and y appear as arguments to f.

Figure 3.1 presents the formulae of subtyping flow analysis — these are identical to the formulae of simple flow analysis. We introduce *subtype*

Formulae:

$$\text{Bool}\ \frac{}{\text{Bool}^{\ell} \in \mathcal{K}^{\leq}(\text{Bool})}$$

$$\to\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \kappa' \in \mathcal{K}^{\leq}(t')}{\kappa \to^{\ell} \kappa' \in \mathcal{K}^{\leq}(t \to t')} \qquad \times\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \kappa' \in \mathcal{K}^{\leq}(t')}{\kappa \times^{\ell} \kappa' \in \mathcal{K}^{\leq}(t \times t')}$$

Subtype relation:

$$\text{Bool}\ \frac{C \vdash \ell_1 \subseteq \ell_2}{C \vdash^{\leq} \text{Bool}^{\ell_1} \leq \text{Bool}^{\ell_2}}$$

$$\text{Arrow}\ \frac{C \vdash^{\leq} \kappa_1 \leq \kappa_1' \quad C \vdash^{\leq} \kappa_2 \leq \kappa_2' \quad C \vdash \ell_1 \subseteq \ell_2}{C \vdash^{\leq} \kappa_1' \to^{\ell_1} \kappa_2 \leq \kappa_1 \to^{\ell_2} \kappa_2'}$$

$$\text{Product}\ \frac{C \vdash^{\leq} \kappa_1 \leq \kappa_1' \quad C \vdash^{\leq} \kappa_2 \leq \kappa_2' \quad C \vdash \ell_1 \subseteq \ell_2}{C \vdash^{\leq} \kappa_1 \times^{\ell_1} \kappa_2 \leq \kappa_1' \times^{\ell_2} \kappa_2'}$$

Figure 3.1: Subtyping flow analysis — formulae and subtype relation

judgements: $C \vdash^{\leq} \kappa \leq \kappa'$ is read "under assumption C, κ is a *subtype* of κ'".

Subtype judgements can be seen as the natural generalisation of the logic of properties to a logic of annotated types. A boolean flow formula Bool^{ℓ_1} is a subtype of another boolean flow formula Bool^{ℓ_2} if we can prove $\ell_1 \subseteq \ell_2$ under assumptions C. The relation is lifted to structured types in the usual manner (contravariant in the argument position for function types) where the annotations on the $\rightarrow$ and $\times$ type constructors are allowed to be subsets just as for booleans.

The subtype relation $\leq$ is transitive as a straightforward consequence of the transitivity of the $\subseteq$ relation.

Figure 3.2 presents the non-logical and semi-logical rules for subtyping flow analysis. The non-logical rules resemble the rules of the simple system; the main difference is that we do not need the subtype step in constructor rules (such as a premise $C \vdash \{l\} \subseteq \alpha$ and conclusion $C; A \vdash^{\leq} \text{True}^l : \text{Bool}^\alpha$ in the Bool-I rule). The rules of the simple system are all derivable. We add a new semi-logical rule called subsumption which allows an expression of type κ to have type κ' if κ is a subtype of κ'.

3.1.1 Subtyping Flow Functions

We define flow function as in chapter 2. Let $\mathcal{T}$ be a derivation. For every l we let $\mathcal{F}_{\mathcal{T}}(l)$ be the least property such that whenever on of the rules

$$\rightarrow\text{-E}\ \frac{C; A \vdash^{\leq} e : \kappa' \rightarrow^{\ell} \kappa \quad C; A \vdash^{\leq} e' : \kappa'}{C; A \vdash^{\leq} e @^l e' : \kappa}$$

$$\times\text{-E}\ \frac{C; A \vdash^{\leq} e : \kappa \times^{\ell} \kappa' \quad C; A, x : \kappa, y : \kappa' \vdash^{\leq} e' : \kappa''}{C; A \vdash^{\leq} \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{Bool-E}\ \frac{C; A \vdash^{\leq} e : \text{Bool}^{\ell} \quad C; A \vdash^{\leq} e' : \kappa \quad C; A \vdash^{\leq} e'' : \kappa}{C; A \vdash^{\leq} \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

is an inference in $\mathcal{T}$ then $C \vdash \ell \leq \mathcal{F}_{\mathcal{T}}(l)$.

3.2 Principality and Minimality

In this section we prove that the subtype system has principal types. We do this in a couple of steps. We give a syntax-directed version of our subtyping flow analysis and then give an algorithm for computing principal types (and we prove that they are indeed principal). We go on to find a minimal, principal typing.

Non-logical rules:

$$\text{Id}\ \frac{}{C; A, x : \kappa \vdash^{\leq} x : \kappa}$$

$$\rightarrow\text{-I}\ \frac{C; A, x : \kappa \vdash^{\leq} e : \kappa'}{C; A \vdash^{\leq} \lambda^{l} x.e : \kappa \rightarrow^{\{l\}} \kappa'}$$

$$\rightarrow\text{-E}\ \frac{C; A \vdash^{\leq} e : \kappa' \rightarrow^{\ell} \kappa \quad C; A \vdash^{\leq} e' : \kappa'}{C; A \vdash^{\leq} e@^{l} e' : \kappa}$$

$$\text{Bool-I}\ \frac{}{C; A \vdash^{\leq} \text{True}^{l} : \text{Bool}^{\{l\}}} \qquad \frac{}{C; A \vdash^{\leq} \text{False}^{l} : \text{Bool}^{\{l\}}}$$

$$\text{Bool-E}\ \frac{C; A \vdash^{\leq} e : \text{Bool}^{\ell} \quad C; A \vdash^{\leq} e' : \kappa \quad C; A \vdash^{\leq} e'' : \kappa}{C; A \vdash^{\leq} \text{if}^{l}\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

$$\times\text{-I}\ \frac{C; A \vdash^{\leq} e : \kappa \quad C; A \vdash^{\leq} e' : \kappa'}{C; A \vdash^{\leq} (e, e')^{l} : \kappa \times^{\{l\}} \kappa'}$$

$$\times\text{-E}\ \frac{C; A \vdash^{\leq} e : \kappa \times^{\ell} \kappa' \quad C; A, x : \kappa, y : \kappa' \vdash^{\leq} e' : \kappa''}{C; A \vdash^{\leq} \text{let}^{l}\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{fix}\ \frac{C; A, x : \kappa \vdash^{\leq} e : \kappa}{C; A \vdash^{\leq} \text{fix}\, x.e : \kappa} \qquad \text{let}\ \frac{C; A \vdash^{\leq} e : \kappa \quad C; A, x : \kappa \vdash^{\leq} e' : \kappa'}{C; A \vdash^{\leq} \text{let}\ x = e\ \text{in}\ e' : \kappa'}$$

Semi-logical rules:

$$\text{Sub}\ \frac{C; A \vdash^{\leq} e : \kappa \quad C \vdash^{\leq} \kappa \leq \kappa'}{C; A \vdash^{\leq} e : \kappa'}$$

Figure 3.2: Subtyping flow analysis — non-logical rules

Non-logical rules:

$$\text{Id}\ \frac{}{C; A, x : \kappa \vdash^{\leq}_{n} x : \kappa}$$

$$\rightarrow\text{-I}\ \frac{C; A, x : \kappa \vdash^{\leq}_{n} e : \kappa'}{C; A \vdash^{\leq}_{n} \lambda^{l} x.e : \kappa \rightarrow^{\{l\}} \kappa'}$$

$$\rightarrow\text{-E}\ \frac{C; A \vdash^{\leq}_{n} e : \kappa' \rightarrow^{\ell} \kappa \quad C; A \vdash^{\leq}_{n} e' : \kappa'' \quad C \vdash^{\leq} \kappa'' \leq \kappa'}{C; A \vdash^{\leq}_{n} e @^{l} e' : \kappa}$$

$$\text{Bool-I}\ \frac{}{C; A \vdash^{\leq}_{n} \text{True}^{l} : \text{Bool}^{\{l\}}} \qquad \frac{}{C; A \vdash^{\leq}_{n} \text{False}^{l} : \text{Bool}^{\{l\}}}$$

$$\text{Bool-E}\ \frac{\begin{array}{ll} C; A \vdash^{\leq}_{n} e : \text{Bool}^{\ell} & C; A \vdash^{\leq}_{n} e' : \kappa' \quad C \vdash^{\leq} \kappa' \leq \kappa \\ & C; A \vdash^{\leq}_{n} e'' : \kappa'' \quad C \vdash^{\leq} \kappa'' \leq \kappa \end{array}}{C; A \vdash^{\leq}_{n} \text{if}^{l}\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

$$\times\text{-I}\ \frac{C; A \vdash^{\leq}_{n} e : \kappa \quad C; A \vdash^{\leq}_{n} e' : \kappa'}{C; A \vdash^{\leq}_{n} (e, e')^{l} : \kappa \times^{\{l\}} \kappa'}$$

$$\times\text{-E}\ \frac{C; A \vdash^{\leq}_{n} e : \kappa \times^{\ell} \kappa' \quad C; A, x : \kappa, y : \kappa' \vdash^{\leq}_{n} e' : \kappa''}{C; A \vdash^{\leq}_{n} \text{let}^{l}\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{fix}\ \frac{C; A, x : \kappa \vdash^{\leq}_{n} e : \kappa' \quad C \vdash^{\leq} \kappa' \leq \kappa}{C; A \vdash^{\leq}_{n} \text{fix}\, x.e : \kappa'}$$

$$\text{let}\ \frac{C; A \vdash^{\leq}_{n} e : \kappa \quad C; A, x : \kappa \vdash^{\leq}_{n} e' : \kappa'}{C; A \vdash^{\leq}_{n} \text{let}\ x = e\ \text{in}\ e' : \kappa'}$$

Figure 3.3: Syntax-directed subtyping flow analysis

3.2.1 A Syntax Directed System

By transitivity the subsumption rule need never be applied twice to the same term. We can thus integrate this rule into the other rules when necessary, in order to achieve an equivalent but syntax-directed type system.

In figure 3.3 we present this version of our subtyping flow analysis. The subscript $_n$ on $\vdash^{\leq}_{n}$ is for *normalised*. The system is obtained by allowing one subtype step where necessary. The following theorem states soundness and completeness of the syntax directed system w.r.t. the non syntax-directed system of figure 3.2.

Theorem 3.1 *Type system $\vdash^{\leq}_{n}$ is sound and complete w.r.t. $\vdash^{\leq}$:*

Soundness *If $C; A \vdash^{\leq}_{n} e : \kappa$ then $C; A \vdash^{\leq} e : \kappa$*

Completeness *If $C; A \vdash^{\leq} e : \kappa$ then there exists κ' such that $C; A \vdash^{\leq}_{n} e : \kappa'$ and $C \vdash^{\leq} \kappa' \leq \kappa$.*

Proof The proof of soundness proceeds by induction over the derivation of $C; A \vdash^{\leq}_{n} e : \kappa$ and the proof of completeness proceeds by induction over the derivation of $C; A \vdash^{\leq} e : \kappa$

□

3.2.2 Algorithm

We define a partial function called *constraints* mapping a subtype constraint $\kappa \leq \kappa'$ to the least sets of inclusion constraints C such that $C \vdash^{\leq} \kappa \leq \kappa'$. We overload function *constraints* to also work on annotation constraints: either by mapping $\ell \subseteq \ell'$ to the set containing the constraint itself or by failing:

$$
\begin{array}{lcll}
\mathit{constraints}(\alpha \subseteq \alpha') & = & \{\alpha \subseteq \alpha'\} & \\
\mathit{constraints}(L \subseteq \alpha) & = & \{L \subseteq \alpha\} & \\
\mathit{constraints}(L \subseteq L') & = & \{\} & \text{if } L \subseteq L' \\
 & = & \mathit{Fail} & \text{otherwise} \\
\mathit{constraints}(\alpha \subseteq L) & = & \mathit{Fail} &
\end{array}
$$

$$
\begin{array}{l}
\mathit{constraints}(\mathrm{Bool}^{\ell} \leq \mathrm{Bool}^{\ell'}) \;=\; \mathit{constraints}(\ell \subseteq \ell') \\
\mathit{constraints}(\kappa_1 \to^{\ell} \kappa_2 \leq \kappa_1' \to^{\ell'} \kappa_2') \\
\quad = \mathit{constraints}(\kappa_1' \leq \kappa_1) \cup \mathit{constraints}(\kappa_2 \leq \kappa_2') \cup \mathit{constraints}(\ell \subseteq \ell') \\
\mathit{constraints}(\kappa_1 \times^{\ell} \kappa_2 \leq \kappa_1' \times^{\ell'} \kappa_2') \\
\quad = \mathit{constraints}(\kappa_1 \leq \kappa_1') \cup \mathit{constraints}(\kappa_2 \leq \kappa_2') \cup \mathit{constraints}(\ell \subseteq \ell')
\end{array}
$$

where we extend the union operator such that $\mathit{Fail} \cup C = \mathit{Fail}$.

Lemma 3.2 *Let κ and κ' be given. If there exists C' s.t. $C' \vdash^{\leq} \kappa \leq \kappa'$ then $\mathit{constraints}(\kappa \leq \kappa')$ returns C such that*

1. $C \vdash^{\leq} \kappa \leq \kappa'$, *and*

2. $C' \vdash^{\leq} C$.

Proof Induction over the structure of the κ (and κ'). □

We go on to define the instance relation used in this chapter. We write $C'; A' \vdash^{\leq} A$ for

> For all assumptions $x : \kappa$ in A, judgement $C'; A' \vdash^{\leq} x : \kappa$ is provable.

Definition 3.3 *A flow judgement* $C'; A' \vdash^{\leq} e' : \kappa'$ *is* an instance[1] *of* $C; A \vdash^{\leq} e : \kappa$ *if* $e = e'$ *and there exists a substitution* S *such that:*

1. $C' \vdash S(C)$,

2. $C'; A' \vdash^{\leq} S(A)$ *and*

3. $C' \vdash^{\leq} S(\kappa) \leq \kappa'$

If $C; A \vdash^{\leq} e : \kappa$ *is also an instance of* $C'; A' \vdash^{\leq} e' : \kappa'$ *then the two judgements are equivalent.*

This instance relation is stronger (that is, more judgements are related) than the instance relation of the previous chapter, since it allows subtyping steps on bindings in the environment and on the result type.

Proposition 3.4 *Derivable judgements are closed under the instance relation. That is, if* $C; A \vdash^{\leq} e : \kappa$ *is derivable then so is any instance of it.*

Proof By induction over the derivation of $C; A \vdash^{\leq} e : \kappa$. □

Figure 3.4 defines a version of algorithm $\mathcal{W}$ implementing the subtype based flow analysis. Like simple types, $\mathcal{W}(A, e)$ can terminate with failure (if the function *constraints* return *Fail*). We therefore define the notion of *proper* environment, which does not need to be as restrictive as in the simple typed case:

Definition 3.5 *We call* A proper *if for all* $x \in \mathrm{Dom}(A)$ *we have that all annotations occurring negatively in* $A(x)$ *are label variables.*

Lemma 3.6 *If* A *is proper then* $\mathcal{W}(A, e)$ *terminates without failure.*

[1] In the literature, this relation is often called *lazy instance* or *strong instance*.

$\mathcal{W}(A, e) =$ case e of

x :	$(\{\}, A(x))$
$\lambda^l x : t.e'$:	let $\kappa \in \mathcal{K}^{\leq}(t)$ be a flow type with fresh variable annotations let $(C, \kappa') = \mathcal{W}((A, x : \kappa), e')$ in $(C, \kappa \rightarrow^{\{l\}} \kappa')$
$e_1 @^l e_2$:	let $(C, \kappa'' \rightarrow^{\ell} \kappa) = \mathcal{W}(A, e_1)$ let $(C', \kappa') = \mathcal{W}(A, e_2)$ in $(C \cup C' \cup \mathit{constraints}(\kappa' \leq \kappa''), \kappa)$
$\mathrm{True}^l(\mathrm{False}^l)$:	$(\{\}, \mathrm{Bool}^{\{l\}})$
$\mathrm{if}^l\ e_1$ then $(e_2 : t)$ else $(e_3 : t)$:	let $\kappa \in \mathcal{K}^{\leq}(t)$ be a flow type with fresh variable annotations let $(C_1, \mathrm{Bool}^{\ell}) = \mathcal{W}(A, e_1)$ let $(C_2, \kappa_2) = \mathcal{W}(A, e_2)$ let $(C_3, \kappa_3) = \mathcal{W}(A, e_3)$ let $C_4 =$ $\mathit{constraints}(\kappa_2 \leq \kappa) \cup$ $\mathit{constraints}(\kappa_3 \leq \kappa)$ in $(C_1 \cup C_2 \cup C_3 \cup C_4, \kappa)$
$(e_1, e_2)^l$:	let $(C_1, \kappa_1) = \mathcal{W}(A, e_1)$ let $(C_2, \kappa_2) = \mathcal{W}(A, e_2)$ in $(C_1 \cup C_2, \kappa_1 \times^{\{l\}} \kappa_2)$
$\mathrm{let}^l\ (x, y)$ be e_1 in e_2 :	let $(C_1, \kappa_x \times^{\ell} \kappa_y) = \mathcal{W}(A, e_1)$ let $(C_2, \kappa) = \mathcal{W}((A, x : \kappa_x, y : \kappa_y), e_2)$ in $(C_1 \cup C_2, \kappa)$
$\mathrm{fix}\, x : t.e'$:	let $\kappa \in \mathcal{K}^{\leq}(t)$ be a flow type with fresh variable annotations let $(C, \kappa') = \mathcal{W}((A, x : \kappa), e')$ in $(C \cup \mathit{constraints}(\kappa' \leq \kappa), \kappa)$
let $x = e_1$ in e_2 :	let $(C_1, \kappa) = \mathcal{W}(A, e_1)$ let $(C_2, \kappa') = \mathcal{W}((A, x : \kappa), e_2)$ in $(C_1 \cup C_2, \kappa')$

Figure 3.4: Algorithm $\mathcal{W}$

Proof We show by induction over the structure of e that if A is proper then

1. $\mathcal{W}(A, e)$ terminates without failure, and
2. $\mathcal{W}(A, e)$ returns (C, κ) where all annotations occurring negatively in κ are label variables.

□

We now go on to prove the main principality theorem. We need a lemma stating that assumptions can be strengthened.

Lemma 3.7 *For all* $C, C', A, x, e, \kappa, \kappa', \kappa''$

1. *If* $C; A \vdash^{\leq} e : \kappa$ *then also* $C \cup C'; A \vdash^{\leq} e : \kappa$, *and*
2. *If* $C; A, x : \kappa' \vdash^{\leq} e : \kappa$ *and* $C \vdash^{\leq} \kappa'' \leq \kappa'$ *then also* $C, C'; A, x : \kappa'' \vdash^{\leq} e : \kappa$

Proof Simple proof by induction over the derivations of $C; A \vdash^{\leq} e : \kappa$ and $C; A, x : \kappa' \vdash^{\leq} e : \kappa$ respectively.

□

We now go on to prove that $\mathcal{W}$ returns principal typings.

Theorem 3.8 *If* $\mathcal{W}(A, e) = (C, \kappa)$ *then* $C; A \vdash^{\leq} e : \kappa$ *and for any* C', κ' *where* $C'; A \vdash^{\leq} e : \kappa'$ *we have that* $C'; A \vdash^{\leq} e : \kappa'$ *is an instance of* $C; A \vdash^{\leq} e : \kappa$.

Proof We prove the more general property:

> If $\mathcal{W}(A, e) = (C, \kappa)$ then $C; A \vdash^{\leq} e : \kappa$ and for any S, C', κ' where $C'; S(A) \vdash^{\leq} e : \kappa'$ we have that $C'; S(A) \vdash^{\leq} e : \kappa'$ is an instance of $C; A \vdash^{\leq} e : \kappa$.
>
> Furthermore, for any α occurring in A
>
> 1. If α has a negative occurrence in κ then it has a negative occurrence in A.
> 2. If α has a positive occurrence in κ then it has a positive occurrence in A.
> 3. If α occurs on the left-hand side of a constraint in C then α has a positive occurrence in A.
> 4. If α occurs on the right-hand side of a constraint in C then α has a negative occurrence in A.

The proof goes by induction over the structure of e:

"$e = x$" Assume $A = A', x : \kappa$. Then, clearly, $\{\}; A', x : \kappa \vdash^{\leq} x : \kappa$. Furthermore, if α has a positive (negative) occurrence in κ then it has a positive (negative) occurrence in A. Since C is empty, point *3.* and *4.* are trivial.

Let S, C, κ' be given such that $C; S(A', x : \kappa) \vdash^{\leq} x : \kappa'$. By theorem 3.1 there exists κ'' such that $C \vdash^{\leq} \kappa'' \leq \kappa'$ and $C; S(A', x : \kappa) \vdash^{\leq}_{n} x : \kappa''$ which in turn implies that $\kappa'' = S(\kappa)$. Now

1. $C \vdash S(\{\})$
2. $C; S(A', x : \kappa) \vdash^{\leq} S(A', x : \kappa)$
3. $C \vdash^{\leq} S(\kappa) \leq \kappa'$

"$e = \lambda^l x.e'$" Let $\kappa \in \mathcal{K}^s(t)$ be a flow type with fresh variable annotations. Let $(C, \kappa') = \mathcal{W}((A, x : \kappa), e')$. By induction

$$C; A, x : \kappa \vdash^{\leq} e' : \kappa'$$

and then

$$C; A \vdash^{\leq} \lambda^l x.e' : \kappa \rightarrow^{\{l\}} \kappa'$$

by the ($\rightarrow$-I) rule.

Let α be a variable occurring in A and $\kappa \rightarrow^{\{l\}} \kappa'$. Since κ was chosen to contain only fresh variables, α must occur in κ' but then points *1.* and *2.* follow by induction. Since C is unchanged, points *3.* and *4.* follow trivially by induction.

Let C', κ_1, S_1 be given such that

$$C'; S_1(A) \vdash^{\leq} \lambda^l x.e' : \kappa_1$$

Then by theorem 3.1 there exists κ_2 such that $C \vdash^{\leq} \kappa_2 \leq \kappa_1$

$$C'; S_1(A) \vdash^{\leq}_{n} \lambda^l x.e' : \kappa_2$$

Hence $\kappa_2 = \kappa_2' \rightarrow^{\{l\}} \kappa_2''$ for some κ_2', κ_2'' and we conclude $\kappa_1 = \kappa_1' \rightarrow^{\ell} \kappa_1''$ for some $\kappa_1', \kappa_1'', \ell$ where $C \vdash^{\leq} \kappa_1' \leq \kappa_2'$, $C \vdash^{\leq} \kappa_2'' \leq \kappa_1''$ and $C \vdash \{l\} \subseteq \ell$.

Furthermore,

$$C'; S_1(A), x : \kappa_2' \vdash^{\leq}_{n} e' : \kappa_2''$$

and by theorem 3.1 also

$$C'; S_1(A), x : \kappa_2' \vdash^{\leq} e' : \kappa_2''$$

and finally, using lemma 3.7

$$C'; S_1(A), x : \kappa_1' \vdash^{\leq} e' : \kappa_2''$$

We chose κ to have fresh variable annotations, and therefore S' exists such that $S'\kappa = \kappa_1'$ and $S'(A(y)) = S_1(A(y))$ for all $y \in \mathrm{Dom}(A)$. Applying the induction hypothesis we find that $C'; S'(A, x : \kappa) \vdash^{\leq} e' : \kappa_2''$ is an instance of $C; A, x : \kappa \vdash^{\leq} e' : \kappa'$ and thus there exists S such that

1. $C' \vdash S(C)$,
2. $C'; S'(A, x : \kappa) \vdash^{\leq} S(A, x : \kappa)$, and
3. $C' \vdash^{\leq} S(\kappa') \leq \kappa_2''$

It is not hard to see that also

1. $C' \vdash S(C)$,
2. $C'; S_1(A) \vdash^{\leq} S(A)$, and
3. $C' \vdash^{\leq} S(\kappa \to^{\{l\}} \kappa') \leq \kappa_1' \to^{\ell} \kappa_1''$

"$e_1@^l e_2$" Let $(C, \kappa'' \to^{\ell} \kappa) = \mathcal{W}(A, e_1)$ and $(C', \kappa') = \mathcal{W}(A, e_2)$. By induction, we find

$$C; A \vdash^{\leq} e_1 : \kappa'' \to^{\ell} \kappa$$

and

$$C'; A \vdash^{\leq} e_2 : \kappa'$$

By lemma 3.7 we find

$$C \cup C' \cup \mathit{constraints}(\kappa' \leq \kappa''); A \vdash^{\leq} e_1 : \kappa'' \to^{\ell} \kappa$$

and

$$C \cup C' \cup \mathit{constraints}(\kappa' \leq \kappa''); A \vdash^{\leq} e_2 : \kappa'$$

and by lemma 3.2

$$C \cup C' \cup \mathit{constraints}(\kappa' \leq \kappa'') \vdash^{\leq} \kappa' \leq \kappa''$$

Thus by the ($\to$-E) rule

$$C \cup C' \cup \mathit{constraints}(\kappa' \leq \kappa''); A \vdash^{\leq} e_1@^l e_2 : \kappa$$

Points *1.* and *2.* follow by induction. Points *3.* and *4.* follow by induction for C and C' and by inspection of $\mathit{constraints}(\kappa' \leq \kappa'')$ using points *1.* and *2.* of the induction hypothesis concerning κ' and κ''.

Let C_1, κ_1, S_1 be given such that

$$C_1; S_1(A) \vdash^{\leq} e_1@^l e_2 : \kappa_1$$

By theorem 3.1 κ_2 exists s.t. $\kappa_2 \leq \kappa_1$ and

$$C_1; S_1(A) \vdash_n^{\leq} e_1@^l e_2 : \kappa_2$$

and since the ($\rightarrow$-E) rule must be the last rule applied, we find that κ_2' exists s.t.

$$C_1; S_1(A) \vdash^{\leq}_{n} e_1 : \kappa_2' \rightarrow^{\ell} \kappa_2$$

and

$$C_1; S_1(A) \vdash^{\leq}_{n} e_2 : \kappa_2'$$

which by theorem 3.1 implies

$$C_1; S_1(A) \vdash^{\leq} e_1 : \kappa_2' \rightarrow^{\ell} \kappa_2 \tag{3.1}$$

and

$$C_1; S_1(A) \vdash^{\leq} e_2 : \kappa_2' \tag{3.2}$$

By the induction hypothesis we find that (3.1) is an instance of $C; A \vdash^{\leq} e_1 : \kappa'' \rightarrow^{\ell} \kappa$ that is there exists S s.t.

1. $C_1 \vdash S(C)$,
2. $C_1; S_1(A) \vdash^{\leq} S(A)$, and
3. $C_1 \vdash^{\leq} S(\kappa'' \rightarrow^{\ell} \kappa) \leq \kappa_2' \rightarrow^{\ell} \kappa_2$

and similarly that (3.2) is an instance of $C'; A \vdash^{\leq} e_2 : \kappa'$ that is there exists S' s.t.

1. $C_1 \vdash S'(C')$,
2. $C_1; S_1(A) \vdash^{\leq} S'(A)$, and
3. $C_1 \vdash^{\leq} S'(\kappa') \leq \kappa_2'$

Without loss of generality, we can assume that $\mathrm{Dom}(S) \cap \mathrm{Dom}(S') \subseteq \mathrm{FlowVar}(A)$. Let α be a variable occurring negatively in A, then we can conclude from the two point 2.'s that

$$C_1 \vdash S\alpha \subseteq S_1\alpha \text{ and } C_1 \vdash S'\alpha \subseteq S_1\alpha \tag{3.3}$$

Similarly, if α occurs positively in A then

$$C_1 \vdash S_1\alpha \subseteq S\alpha \text{ and } C_1 \vdash S_1\alpha \subseteq S'\alpha \tag{3.4}$$

Thus, if α occurs both negatively and positively then $S\alpha = S'\alpha$.

Now define S'' as follows:

$$S''(\alpha) = \begin{cases} S_1(\alpha) & \text{, if } \alpha \in \mathrm{Dom}(S) \cap \mathrm{Dom}(S') \\ S(\alpha) & \text{, if } \alpha \in \mathrm{Dom}(S) \setminus \mathrm{Dom}(S') \\ S'(\alpha) & \text{, if } \alpha \in \mathrm{Dom}(S') \setminus \mathrm{Dom}(S) \end{cases}$$

Let $\alpha \subseteq \beta$ be a constraint in C. We know $C_1 \vdash S(\alpha \subseteq \beta)$ and want to prove that $C_1 \vdash S''(\alpha \subseteq \beta)$. Four cases:

1. If $\alpha, \beta \in \mathrm{Dom}(S) \setminus \mathrm{Dom}(S')$ then $S''(\alpha \subseteq \beta) = S(\alpha \subseteq \beta)$ and the result follows immediately.
2. If $\alpha, \beta \in \mathrm{Dom}(S) \cap \mathrm{Dom}(S')$ then $S''(\alpha \subseteq \beta) = S_1(\alpha \subseteq \beta)$. By the induction hypothesis, α occurs positively and β occurs negatively in A. Thus by (3.3) and (3.4) we have $C_1 \vdash S_1\alpha \subseteq S\alpha$ and $C_1 \vdash S\beta \subseteq S_1\beta$. The result follows by transitivity.
3. If $\alpha \in \mathrm{Dom}(S) \cap \mathrm{Dom}(S')$ and $\beta \in \mathrm{Dom}(S) \setminus \mathrm{Dom}(S')$ then the result follows by the same reasoning as in points 1. and 2.
4. If $\beta \in \mathrm{Dom}(S) \cap \mathrm{Dom}(S')$ and $\alpha \in \mathrm{Dom}(S) \setminus \mathrm{Dom}(S')$ then the result follows by the same reasoning as in points 1. and 2.

By a similar method, we can prove that if $L \subseteq \alpha$ is a constraint in C then $C_1 \vdash S''(L \subseteq \alpha)$. We can therefore conclude $C_1 \vdash S''C$. We prove $C_1 \vdash S''C'$ in the same way.

We prove $C_1 \vdash^{\leq} S''(\kappa' \leq \kappa'')$ from which $C_1 \vdash S''(\mathit{constraints}(\kappa' \leq \kappa''))$ follows by lemma 3.2. We know that $C_1 \vdash^{\leq} S'\kappa' \leq \kappa_2'$ and $C_1 \vdash^{\leq} \kappa_2' \leq S\kappa''$. Thus the result follows by transitivity if we can prove

1. $C_1 \vdash^{\leq} S''\kappa' \leq S'\kappa'$, and
2. $C_1 \vdash^{\leq} S\kappa'' \leq S''\kappa''$

The proofs go as follows:

1. If $\alpha \notin \mathrm{Dom}(S') \cap \mathrm{Dom}(S)$ we are trivially done, so assume $\alpha \in \mathrm{Dom}(S') \cap \mathrm{Dom}(S)$. If α occurs positively in κ' then it also occurs positively in A and by (3.4) we have $C_1 \vdash^{\leq} S_1\alpha \subseteq S'\alpha$. Similarly if α occurs negatively in κ' we find by (3.3) that $C_1 \vdash^{\leq} S'\alpha \subseteq S_1\alpha$.
2. Similarly assume that $\alpha \in \mathrm{Dom}(S') \cap \mathrm{Dom}(S)$. If α occurs positively in κ'' then it occurs negatively $\kappa'' \to^{\ell} \kappa$ and therefore negatively in A. Thus by (3.3) that $C_1 \vdash^{\leq} S\alpha \subseteq S_1\alpha$. Similarly, if it occurs negatively in κ'' we find by (3.4) that $C_1 \vdash^{\leq} S_1\alpha \subseteq S\alpha$.

We have thus proven $C_1 \vdash S''(C \cup C' \cup \mathit{constraints}(\kappa' \leq \kappa''))$.

The second property we need to prove is $C_1; S_1(A) \vdash^{\leq} S''(A)$ which follows easily using (3.3) and (3.4).

Finally, we need to prove $C_1 \vdash^{\leq} S''\kappa \leq \kappa_2$. We know that $C_1 \vdash^{\leq} S\kappa \leq \kappa_2$ so we just need to show $C_1 \vdash^{\leq} S''\kappa \leq S\kappa$ which follows in the same way as the proof of $C_1 \vdash^{\leq} S''\kappa' \leq S'\kappa'$ above.

We can summarize

1. $C_1 \vdash^{\leq} S''(C \cup C' \cup \mathit{constraints}(\kappa' \leq \kappa''))$,
2. $C_1; S_1(A) \vdash^{\leq} S''(A)$, and

3. $C_1 \vdash^{\leq} S''(\kappa) \leq \kappa_2$

which is exactly what we need to prove that $C_1; S_1(A) \vdash^{\leq} e_1@^l e_2 : \kappa_1$ is an instance of $C \cup C' \cup \mathit{constraints}(\kappa' \leq \kappa''); A \vdash^{\leq} e_1@^l e_2 : \kappa$.

The remaining cases are similar.

□

The above lemma and theorem show that every term has a principal type under proper environments.

3.2.3 Minimality

The approach to minimality taken in chapter 2 works here as well though it takes a little more effort to remove variables not occurring in the type or environment. We are, however, able to do a bit better: the lazy part (allowing a subtype step on the type in principal typings) makes it possible to remove variables occurring in A or κ.

Definition 3.9 *Assume $A = \{x_1 : \kappa_1, \cdots, x_n : \kappa_n\}$. A flow variable occurring free in A or κ occurs* positively *(*negatively*) in $C; A \vdash e : \kappa$ if it occurs positively (negatively) in $\kappa_1 \to \cdots \to \kappa_n \to \kappa$. Variables not occurring free in A or κ are said to be* neutral.

All variables occurring in C will occur positively, negatively, positively and negatively, or be neutral in $C; A \vdash e : \kappa$. We would like to remove all neutral variables from C as they do no contribute to the flow information of the judgement. Alternatively, we can think of neutral variables as describing flow in a subexpression of e which has played its part; hence their flow value can be resolved without jeopardising principality. Neutral variables *do* play a role in the final judgement by, via transitivity, being responsible for inequalities between non-neutral variables. E.g. assume that α_i is non-neutral for all i and β is neutral in

$$C = \{\alpha_1 \subseteq \beta, \alpha_2 \subseteq \beta, \beta \subseteq \alpha_3, \beta \subseteq \alpha_4\}$$

Clearly, we cannot simply remove β (and all constraints involving it). We have to represent the inequalities in C without using β. This can be done as:

$$C' = \{\alpha_1 \subseteq \alpha_3, \alpha_1 \subseteq \alpha_4, \alpha_2 \subseteq \alpha_3, \alpha_2 \subseteq \alpha_4\}$$

If we assume that $C; A \vdash^{\leq} e : \kappa$ is principal, we wish the resulting judgement $C'; A \vdash^{\leq} e : \kappa$ to be principal as well. This is ensured by proving the two judgements equivalent (lazy instances of each other). This is, however, *not* possible for the above judgements: there exists no substitution S such that $C' \vdash S(C)$. Here, the introduction of $\sqcup$ into the language of annotations prove important: we can simply map β to $\alpha_1 \sqcup \alpha_2$.

Due to the identity rule (Id) we can assume that constraint sets contain no constraints of the form $\alpha \leq \alpha$. To be more precise, if C' is the result of removing all constraints of the form $\alpha \leq \alpha$ from a constraint set C, then $C; A \vdash^{\leq} e : \kappa$ and $C'; A \vdash^{\leq} e : \kappa$ are equivalent.

Lemma 3.10 *Let α be a flow variable with no negative occurrences in $C; A \vdash^{\leq} e : \kappa$.*

Let $\ell_1 \subseteq \alpha, \cdots, \ell_n \subseteq \alpha$ be all inequalities in C with α on the right-hand side. We delete these from C and replace every inequality $\alpha \subseteq \beta$ in C by $\ell_1 \subseteq \beta, \cdots, \ell_n \subseteq \beta$ and call the resulting constraint set C'. Let $S = \{\bigsqcup_i \ell_i / \alpha\}$.

Then $C; A \vdash^{\leq} e : \kappa$ and $C'; S(A) \vdash^{\leq} e : S(\kappa)$ are equivalent (lazy instances of each other).

Proof We remark that $C \vdash \bigsqcup_i \ell_i \subseteq \alpha$. First, $C; A \vdash^{\leq} e : \kappa$ is an instance of $C'; S(A) \vdash^{\leq} e : S(\kappa)$ since

1. $C \vdash C'$,
2. $C; A \vdash^{\leq} S(A)$, and
3. $C \vdash^{\leq} S(\kappa) \leq \kappa$

where 1. is proven by transitivity and 2. and 3. are proven using that α occurs negatively in A and positively in κ (if it occurs at all).

To see that $C'; S(A) \vdash^{\leq} e : S(\kappa)$ is an instance of $C; A \vdash^{\leq} e : \kappa$, we see

1. $C' \vdash S(C)$,
2. $C'; S(A) \vdash^{\leq} S(A)$, and
3. $C' \vdash^{\leq} S(\kappa) \leq S(\kappa)$

where 1. follows from $C' \vdash^{\leq} \ell_j \leq \bigsqcup_i \ell_i$ for all j and $C' \vdash^{\leq} \bigsqcup_i \ell_i \leq \beta$. Points 2. and 3. are trivial.

□

Constraints $L \subseteq \alpha$ and $L' \subseteq \alpha$ can be replaced by $L \cup L' \subseteq \alpha$. The following lemma shows this to lead to an equivalent judgement:

Lemma 3.11 *Let α be any flow variable. Then $C, L \subseteq \alpha, L' \subseteq \alpha; A \vdash^{\leq} e : \kappa$ and $C, L \cup L' \subseteq \alpha; A \vdash^{\leq} e : \kappa$ are equivalent.*

By applying lemma 3.10 to all neutral variables α in a judgement and applying lemma 3.11 exhaustively, we arrive at the same theorem as in chapter 2:

Theorem 3.12 *For every flow judgement $C; A \vdash e : \kappa$ there exists an equivalent judgement $C'; A \vdash e : \kappa$ where C' contains only flow variables α occurring free in A or κ and for each α there is only one constraint of the form $L \subseteq \alpha$ in C'.*

Note, that according to lemma 3.10, we can remove more variables than this (namely those occurring negatively), but by applying the lemma to neutral variables, we arrive at a judgement without $\sqcup$ (because the substitution S has no effect on A and κ).

Let $\mathcal{T}$ be a subtyping flow derivation. If we apply the substitution S computed by lemmas 3.10 and 3.11 to $\mathcal{F}_\mathcal{T}$ we get a new flow function $S(\mathcal{F}_\mathcal{T})$ with the special property that all destructors are mapped to expressions $L \sqcup \bigsqcup_i \alpha_i$ over flow variables free in A or κ — thus anything that can be ground (without loosing modularity) will be ground in $\mathcal{F}_\mathcal{T}$.

3.3 Sestoft's Closure Analysis

In this section we present the closure analysis developed by Sestoft in [Ses88, Ses91]. The analysis is based on abstract interpretation. We will try to be faithful to Sestoft's presentation of the analysis, but we have to adapt it to our language and to extend it to handle flow of other data than functions (closures).

For the analysis we will use two functions ϕ and ρ mapping labels to sets of labels. Their intended meaning is as follows:

$$\begin{array}{rcl} \phi l & = & \begin{cases} \text{the set of labels that the body } e \text{ of } \lambda^l x.e \text{ can evaluate to, or} \\ \text{a pair of sets of labels that the subexpression } e_1, e_2 \text{ of} \\ (e_1, e_2)^l \text{ can evaluate to} \end{cases} \\ \rho x & = & \text{the set of labels that } x \text{ can evaluate to.} \end{array}$$

The two analysis functions $\mathcal{P}_e$ (the *closure analysis function*) and $\mathcal{P}_v$ (the *closure propagation function*) have the following meanings:

$$\begin{array}{rcl} \mathcal{P}_e[\![e]\!]\phi\rho & = & \text{the set of labels that expression } e \text{ can evaluate to.} \\ \mathcal{P}_v[\![e]\!]\phi\rho x & = & \text{the set of labels that } x \text{ can evaluate to in } e. \end{array}$$

(remember, that we are assuming that all variable names are distinct.)

The functions are defined as follows (where we assume that $x \neq y \neq z \neq x$):

$$
\begin{array}{rcl}
\mathcal{P}_e[\![x]\!]\phi\rho & = & \rho(x) \\
\mathcal{P}_e[\![\lambda^l x.e]\!]\phi\rho & = & \{l\} \\
\mathcal{P}_e[\![e_1 @^l e_2]\!]\phi\rho & = & \bigcup\{\phi l' \mid l' \in \mathcal{P}_e[\![e_1]\!]\phi\rho\} \\
\mathcal{P}_e[\![\mathrm{True}^l]\!]\phi\rho & = & \{l\} \\
\mathcal{P}_e[\![\mathrm{False}^l]\!]\phi\rho & = & \{l\} \\
\mathcal{P}_e[\![\mathrm{if}^l\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3]\!]\phi\rho & = & \mathcal{P}_e[\![e_2]\!]\phi\rho \cup \mathcal{P}_e[\![e_3]\!]\phi\rho \\
\mathcal{P}_e[\![(e_1, e_2)^l]\!]\phi\rho & = & \{l\} \\
\mathcal{P}_e[\![\mathrm{let}^l\ (x,y)\ \mathrm{be}\ e_1\ \mathrm{in}\ e_2]\!]\phi\rho & = & \mathcal{P}_e[\![e_2]\!]\phi\rho \\
\mathcal{P}_e[\![\mathrm{fix}^l x.e]\!]\phi\rho & = & \mathcal{P}_e[\![e]\!]\phi\rho \\
\mathcal{P}_e[\![\mathrm{let}^l\ x = e_1\ \mathrm{in}\ e_2]\!]\phi\rho & = & \mathcal{P}_e[\![e_2]\!]\phi\rho
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{P}_v[\![x]\!]\phi\rho y & = & \{\} \\
\mathcal{P}_v[\![y]\!]\phi\rho y & = & \{\} \\
\mathcal{P}_v[\![\lambda^l x.e]\!]\phi\rho y & = & \mathcal{P}_v[\![e]\!]\phi\rho y \\
\mathcal{P}_v[\![\lambda^l y.e]\!]\phi\rho y & = & \mathcal{P}_v[\![e]\!]\phi\rho y \\
\mathcal{P}_v[\![e_1 @^l e_2]\!]\phi\rho y & = & \mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y \\
 & & \cup \bigcup\{\mathcal{P}_e[\![e_2]\!]\phi\rho \mid y \text{ is bound by } \lambda^l y.e' \\
 & & \qquad \text{where } l \in \mathcal{P}_e[\![e_1]\!]\phi\rho\} \\
\mathcal{P}_v[\![\mathrm{True}^l]\!]\phi\rho y & = & \{\} \\
\mathcal{P}_v[\![\mathrm{False}^l]\!]\phi\rho y & = & \{\} \\
\mathcal{P}_v[\![\mathrm{if}^l\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3]\!]\phi\rho y & = & \mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y \cup \mathcal{P}_v[\![e_3]\!]\phi\rho y \\
\mathcal{P}_v[\![(e_1, e_2)^l]\!]\phi\rho y & = & \mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y \\
\mathcal{P}_v[\![\mathrm{let}^l\ (x,z)\ \mathrm{be}\ e_1\ \mathrm{in}\ e_2]\!]\phi\rho y & = & \mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y \\
\mathcal{P}_v[\![\mathrm{let}^l\ (y,x)\ \mathrm{be}\ e_1\ \mathrm{in}\ e_2]\!]\phi\rho y & = & \mathit{fst}(\phi(\mathcal{P}_e[\![e_1]\!]\phi\rho)) \cup \mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y \\
\mathcal{P}_v[\![\mathrm{let}^l\ (x,y)\ \mathrm{be}\ e_1\ \mathrm{in}\ e_2]\!]\phi\rho y & = & \mathit{snd}(\phi(\mathcal{P}_e[\![e_1]\!]\phi\rho)) \cup \mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y \\
\mathcal{P}_v[\![\mathrm{fix}\, x.e]\!]\phi\rho y & = & \mathcal{P}_v[\![e]\!]\phi\rho y \\
\mathcal{P}_v[\![\mathrm{fix}\, y.e]\!]\phi\rho y & = & \mathcal{P}_v[\![e]\!]\phi\rho y \cup \mathcal{P}_e[\![e]\!]\phi\rho \\
\mathcal{P}_v[\![\mathrm{let}\ y = e_1\ \mathrm{in}\ e_2]\!]\phi\rho y & = & \mathcal{P}_e[\![e_1]\!]\phi\rho \cup \mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y \\
\mathcal{P}_v[\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e_2]\!]\phi\rho y & = & \mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y
\end{array}
$$

We seek descriptions ϕ, ρ which constitute the least simultaneous solution to the equations:

$$
\begin{array}{rcll}
\phi l & = & \mathcal{P}_e[\![e_1]\!]\phi\rho & \text{, for all } \lambda^l x.e_1 \text{ in } e \\
\phi l & = & (\mathcal{P}_e[\![e_1]\!]\phi\rho, \mathcal{P}_e[\![e_2]\!]\phi\rho) & \text{, for all } (e_1, e_2)^l \text{ in } e \\
\rho x & = & \mathcal{P}_v[\![e]\!]\phi\rho x &
\end{array}
$$

3.4 Constraint Based Analysis

Palsberg defines flow analysis by first generating a number of constraints from the syntax tree of the analysed term and then solving these [Pal94]. The presentation in this section borrows some notation from [PO95] (and is the same as in section 2.4) and extends the analysis to our richer language.

Let X_e and Y_e be as in section 2.4. The constraint generation is shown in figure 3.5. It generates constraints over $X_e \cup Y_e$.

Generating constraints for e.

For every occurrence in e of:	generate:
x	$x \subseteq [\![x]\!]$
$\lambda^l x.e_1$	$\{l\} \subseteq [\![\lambda^l x.e_1]\!]$
$e_1 @ e_2$	for every $\lambda^l x.e_3$ in e:
	$\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_2]\!] \subseteq x$
	$\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_3]\!] \subseteq [\![e_1 @ e_2]\!]$
True^l	$\{l\} \subseteq [\![\text{True}^l]\!]$
False^l	$\{l\} \subseteq [\![\text{False}^l]\!]$
if e_1 then e_2 else e_3	$[\![e_2]\!] \subseteq [\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]$
	$[\![e_3]\!] \subseteq [\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!]$
$(e_1, e_2)^l$	$\{l\} \subseteq [\![(e_1, e_2)^l]\!]$
let (x, y) be e_1 in e_2	$[\![e_2]\!] \subseteq [\![\text{let } (x, y) \text{ be } e_1 \text{ in } e_2]\!]$
	for every $(e_3, e_4)^l$ in e:
	$\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_3]\!] \subseteq x$
	$\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_4]\!] \subseteq y$
$\text{fix}\, x.e_1$	$[\![e_1]\!] \subseteq x$
	$[\![e_1]\!] \subseteq [\![\text{fix}\, x.e_1]\!]$
let $x = e_1$ in e_2	$[\![e_1]\!] \subseteq x$
	$[\![e_2]\!] \subseteq [\![\text{let } x = e_1 \text{ in } e_2]\!]$

Figure 3.5: Constraint generation a la Palsberg

3.5 Equivalences

This section proves the equivalence between the three flow analyses defined in this chapter (since the equivalences are known, this amounts to recapitulating other people's proofs). This is done only to relate my work to previous work in flow analysis, and we will not be very verbose in the proofs. The equivalence does give us preservation of flow information under arbitrary β-reduction for free, as this was proven by Palsberg for constraint based flow analysis [Pal94].

3.5.1 Equivalence between Constraint Based Analysis and Closure Analysis

The equivalence between closure analysis and constraint based analysis was proven by Palsberg in [Pal94]. Though his proof was for Bondorf's variant

of closure analysis [Bon91] and we have changed the language slightly, we see no reason to repeat Palsberg's proof in detail.

We will briefly sketch the equivalence to help the reader's intuition. What we will do is transform closure analysis to constraint form and realise that the constraints generated are the same as Palsberg's. Given ϕ, ρ we introduce new notation for closure analysis

$$\begin{array}{rcl} [\![e]\!] & = & \mathcal{P}_e[\![e]\!]\phi\rho \\ \langle\!| x |\!\rangle & = & \mathcal{P}_v[\![p]\!]\phi\rho x \end{array}$$

where p is the whole program. Now, ϕ, ρ are solutions to the equations of section 3.3 if and only if the equations generated as follows are true:

Case e of

x	:	$[\![e]\!] = \langle\!| x |\!\rangle$
$\lambda^l x.e_1$	:	$[\![e]\!] = \{l\}$
$e_1 @^l e_2$	:	$[\![e]\!] = \bigcup\{\phi l' \mid l' \in [\![e_1]\!]\}$
True^l	:	$[\![e]\!] = \{l\}$
False^l	:	$[\![e]\!] = \{l\}$
if e_1 then e_2 else e_3	:	$[\![e]\!] = [\![e_2]\!] \cup [\![e_3]\!]$
$(e_1, e_2)^l$	:	$[\![e]\!] = \{l\}$
let (x, y) be e_1 in e_2	:	$[\![e]\!] = [\![e_2]\!]$
fix $x.e_1$	:	$[\![e]\!] = [\![e_1]\!]$
let $x = e_1$ in e_2	:	$[\![e]\!] = [\![e_2]\!]$

Case e of

x	:	$\{\}$
$\lambda^l x.e_1$	:	$\{\}$
$e_1 @^l e_2$	:	$\{[\![e_2]\!] \subseteq \langle\!| x |\!\rangle \mid x$ is bound by $\lambda^l x.e'$ where $l \in [\![e_1]\!]\}$
True^l	:	$\{\}$
False^l	:	$\{\}$
if e_1 then e_2 else e_3	:	$\{\}$
$(e_1, e_2)^l$	:	$\{\}$
let (x, y) be e_1 in e_2	:	$\mathit{fst}(\phi([\![e_1]\!])) \subseteq \langle\!| x |\!\rangle$ and $\mathit{snd}(\phi([\![e_1]\!])) \subseteq \langle\!| y |\!\rangle$
fix $x.e_1$	:	$\langle\!| x |\!\rangle \supseteq [\![e_1]\!]$
let $x = e_1$ in e_2	:	$\langle\!| x |\!\rangle \supseteq [\![e_1]\!]$

The first case corresponds to the definition of $\mathcal{P}_e$ and the second to the definition of $\mathcal{P}_v$. It is easy to see the first case arises simply from the change of syntax above: eg. $[\![e]\!] = [\![e_2]\!] \cup [\![e_3]\!]$ is simply shorthand notation for the equation $\mathcal{P}_e[\![\text{if}^l\ e_1 \text{ then } e_2 \text{ else } e_3]\!]\phi\rho = \mathcal{P}_e[\![e_2]\!]\phi\rho \cup \mathcal{P}_e[\![e_3]\!]\phi\rho$.

The second case is a bit more tricky: the definition of $\mathcal{P}_v$ computes the union of possible bindings of one variable at a time. We want to generate equations for all variables at once.

Let us examine in detail the application case where

$$\begin{aligned}\mathcal{P}_v[\![e_1 @^l e_2]\!]\phi\rho y \quad = \quad & \mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y \\ & \cup \bigcup\{\mathcal{P}_e[\![e_2]\!]\phi\rho \mid y \text{ is bound by } \lambda^l y.e' \\ & \qquad \text{where } l \in \mathcal{P}_e[\![e_1]\!]\phi\rho\}\end{aligned}$$

First, note that we are generating equations resulting from the application itself, so the first components $\mathcal{P}_v[\![e_1]\!]\phi\rho y \cup \mathcal{P}_v[\![e_2]\!]\phi\rho y$ can be dropped. We are left with

$$\bigcup\{\mathcal{P}_e[\![e_2]\!]\phi\rho \mid y \text{ is bound by } \lambda^l y.e' \text{ where } l \in \mathcal{P}_e[\![e_1]\!]\phi\rho\}$$

which by our shorthand notation is

$$\bigcup\{[\![e_2]\!] \mid y \text{ is bound by } \lambda^l y.e' \text{ where } l \in [\![e_1]\!]\}$$

This is the total contribution for y, so we generate the constraints

$$\{[\![e_2]\!] \subseteq \langle\!| y |\!\rangle \mid y \text{ is bound by } \lambda^l y.e' \text{ where } l \in [\![e_1]\!]\}$$

We still have ϕ occurring in a few places:

1. In the first definition, the application case generates:

$$[\![e]\!] = \bigcup\{\phi l' \mid l' \in [\![e_1]\!]\}$$

   We can obviously replace this with:

$$\bigcup\{[\![e']\!] \mid \lambda^l y.e' \text{ is in } p \text{ and } l \in [\![e_1]\!]\} = [\![e]\!]$$

2. In the pair-destructing case of the second definition:

$$\mathit{fst}(\phi([\![e_1]\!])) \subseteq \langle\!| x |\!\rangle \text{ and } \mathit{snd}(\phi([\![e_1]\!])) \subseteq \langle\!| y |\!\rangle$$

   This can be replaced by

$$\begin{aligned}&\text{for every } (e_3, e_4)^l \text{ in } e\text{:} \\ &\quad \{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_3]\!] \subseteq \langle\!| x |\!\rangle \\ &\quad \{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_4]\!] \subseteq \langle\!| y |\!\rangle\end{aligned}$$

Instead of viewing $[\![e]\!]$ and $\langle\!| x |\!\rangle$ as shorthand notation, we can view them as *variable names*. The equations can then be seen as constraints over variables to be solved.

Clearly the two definitions generating constraints can be merged:

Case e of

x	:	$[\![e]\!] = \langle\!\langle\, x \,\rangle\!\rangle$
$\lambda^l x.e_1$	:	$[\![e]\!] = \{l\}$
$e_1 @ e_2$	:	$\bigcup\{[\![e']\!] \mid \lambda^l y.e' \text{ is in } p \text{ and } l \in [\![e_1]\!]\} = [\![e]\!]$ and $\bigcup\{[\![e_2]\!] \mid x \text{ is bound by } \lambda^l y.e' \text{ where } l \in [\![e_1]\!]\} \subseteq \langle\!\langle\, x \,\rangle\!\rangle$
True^l	:	$[\![e]\!] = \{l\}$
False^l	:	$[\![e]\!] = \{l\}$
if e_1 then e_2 else e_3	:	$[\![e]\!] = [\![e_2]\!] \cup [\![e_3]\!]$
$(e_1, e_2)^l$	:	$[\![e]\!] = \{l\}$
let (x, y) be e_1 in e_2	:	$[\![e]\!] = [\![e_2]\!]$ and for every $(e_3, e_4)^l$ in e: $\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_3]\!] \subseteq \langle\!\langle\, x \,\rangle\!\rangle$ $\{l\} \subseteq [\![e_1]\!] \Longrightarrow [\![e_4]\!] \subseteq \langle\!\langle\, y \,\rangle\!\rangle$
fix $x.e_1$	:	$[\![e]\!] = [\![e_1]\!]$ and $\langle\!\langle\, x \,\rangle\!\rangle \supseteq [\![e_1]\!]$
let $x = e_1$ in e_2	:	$[\![e]\!] = [\![e_2]\!]$ and $\langle\!\langle\, x \,\rangle\!\rangle \supseteq [\![e_1]\!]$

Comparing this with the constraint generation of Palsberg (section 3.4) we see a strong resemblance (up to the syntactical difference between $\langle\!\langle\, x \,\rangle\!\rangle$ and x). It is easy to see that any solution to a constraint set generated as above will be a solution to a constraint set generated using Palsberg's system. The other direction is not true, but it holds that for any solution to a constraint set generated by Palsberg's method, there exists a solution to the constraint set generated as above which is smaller: if S is a solution to Palsberg's constraints, there exists a solution S' to the above constraints such that for all variables V, $S'(V) \subseteq S(V)$. In particular the minimal solution to Palsberg's system will be the minimal solution to the above (remember that when we introduced $\subseteq$ constraints instead of unions in the definition of $\mathcal{P}_v$ only the minimal solution to the equations corresponded to Sestoft's analysis).

3.5.2 Equivalence between Subtype and Constraint Based Analysis

Palsberg and O'Keefe [PO95] and Heintze [Hei95] proved the equivalence between the constraint based analysis and a type based analysis like the above. All this, however, was done in an untyped setting so as in the equivalence sketched in section 2.4 recursive types are added. Furthermore, the system includes a top type and a bottom type (Heintze defines $\bot = \mu\alpha.\alpha$ which is not allowed in Amadio and Cardelli's type system [AC91] and hence not in Palsberg and O'Keefe's). Using an argument similar to the argument in section 2.4 we realize that our subtype based analysis has the same precision as the constraint based analysis for well-typed terms (i.e. for typed terms,

constraint based analysis finds a solution equivalent to the minimal ground derivation in our system).

3.5.3 Complexity

Sestoft's closure analysis as well as Palsberg and O'Keefe's constraint based analysis have been proven to be computable in cubic time $\mathcal{O}(n^3)$ where n is the size of the untyped program.

Implementing subtype flow analysis using algorithm $\mathcal{W}$ as described above will not lead to any improvement of this complexity. The algorithm will produce a pair (C, κ) where the size of C is proportional to the size of the typed program. Thus the complexity of this analysis is inherently exponential in the size of the untyped program — assuming that types are bounded, we do arrive at a complexity which is consistent with Sestoft's and Palsberg and O'Keefe's:

Let n be the size of the typed program; then n is proportional to the size of C and to the number of variables in C. Since lemma 3.10 removes a variable at each application, it can be applied at most n times. Each application traverses the whole constraint set: unfortunately we cannot still assume this to be proportional in size to the size of the typed program since the lemma adds new constraints, but it is bounded by n^2. Applying lemma 3.10 overshadows application of lemma 3.11 and hence the complexity of theorem 3.12 is $\mathcal{O}(n^3)$. Careful engineering might be able to keep the size of C linear in the size of the typed program thus reducing the complexity to $\mathcal{O}(n^2)$ — since the next chapter presents a simple and elegant method of achieving this complexity, we will not pursue this further here.

Note that in practical implementations it can be desirable to apply the reduction procedure of lemma 3.10 not only to the final result, but also during analysis in order to keep constraint sets of manageable size.

3.6 Soundness

Preservation under call-by-name and call-by-need reduction has been proven by Sestoft. Invariance under arbitrary β-reduction has been proven by Palsberg.

A direct proof of soundness follows the proof of soundness of simple flow analysis directly — the substitution lemma and subject reduction theorem only differ from the equivalent statements in chapter 2 by allowing for subtyping: the judgements in redex and reduct for expressions with the same label are not required to assign the *same* type, but there should exist a common subtype to all types.

Lemma 3.13 (Substitution lemma) *If* $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C; A, x : \kappa' \vdash^{\leq} e : \kappa}$ *and* $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C; A \vdash^{\leq} e' : \kappa'}$ *then there exists* $\mathcal{T}_3$ *such that*

1. $\mathcal{T}_3 = \dfrac{\mathcal{T}_3}{C; A \vdash^{\leq} e[e'/x] : \kappa}$ *and*
2. *For all* $l \in \text{Destructors}(e[e'/x])$ *we have*

$$C \vdash \mathcal{F}_{\mathcal{T}_3}(l) = \mathcal{F}_{\mathcal{T}_1}(l) \sqcup \mathcal{F}_{\mathcal{T}_2}(l)$$

Proof The lemma follows by induction on the structure of $\mathcal{T}_1$. All cases are similar to the proof of lemma 2.9 except:

(Sub): Assume

$$\mathcal{T}_1 = \frac{\mathcal{T}_1' \quad C \vdash^{\leq} \kappa'' \leq \kappa}{C; A, x : \kappa' \vdash^{\leq} e : \kappa} \quad \text{where} \quad \mathcal{T}_1' = \frac{\mathcal{T}_1''}{C; A, x : \kappa' \vdash^{\leq} e : \kappa''}$$

and

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{C; A \vdash^{\leq} e' : \kappa'}$$

By induction there exists $\mathcal{T}_4$ such that

$$\mathcal{T}_4 = \frac{\mathcal{T}_4'}{C; A \vdash^{s} e[e'/x] : \kappa''}$$

and for all $l \in \text{Destructors}(e[e'/x])$ we have

$$C \vdash \mathcal{F}_{\mathcal{T}_4}(l) = \mathcal{F}_{\mathcal{T}_1''}(l) \sqcup \mathcal{F}_{\mathcal{T}_2}(l)$$

Now

1. $$\mathcal{T}_3 = \frac{\dfrac{\mathcal{T}_4'}{C; A \vdash^{s} e[e'/x] : \kappa''} \quad C \vdash^{\leq} \kappa'' \leq \kappa}{C; A \vdash^{\leq} e[e'/x] : \kappa}$$
2. Follows by noting that $\mathcal{F}_{\mathcal{T}_4}(l) = \mathcal{F}_{\mathcal{T}_3}(l)$ for all l.

□

We then prove the main subject reduction theorem which is also extended with preservation of types of subexpressions.

Theorem 3.14 (Subject Reduction) *If* $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C; A \vdash^{\leq} e_1 : \kappa}$ *and* $e_1 \longrightarrow e_2$ *then there exists* $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C; A \vdash^{\leq} e_2 : \kappa}$ *and*

2. *For all* $l \in \text{Destructors}(e')$ *we have*

$$C \vdash \mathcal{F}_{\mathcal{T}_2}(l) \subseteq \mathcal{F}_{\mathcal{T}_1}(l)$$

and if $l \in \text{Destructors}(e)$ *consumes* $l' \in \text{Contructors}(e)$ *then*

$$C \vdash \{l'\} \subseteq \mathcal{F}_{\mathcal{T}_1}(l)$$

Proof The proof is like the proof of theorem 2.10: the non-trivial cases follow from lemma 3.13. □

Soundness as defined in section 1.7 follows as a trivial corollary (induction on the length of the reduction):

Corollary 3.15 (Soundness for Simple Flow Analysis) *Let* $\mathcal{T}$ *be any derivation for* e *and let* $C; A \vdash^{\leq} e : \kappa$ *be its conclusion. Then* $C; \mathcal{F}_{\mathcal{T}} \models e$.

Chapter 4

Flow Graphs

In the previous chapter, we saw basic flow analysis expressed using constraints, abstract interpretation and an annotated type system. In this chapter we will present *value flow graphs*. The graphs presented in this chapter will have the same accuracy as the analyses of the previous chapter, but will carry even more intensional information: it will be possible to trace the exact path a value follows through the program from creation to destruction.

Untyped graphs can be viewed as a graphical representation of Palsberg's constraint based definition of basic flow analysis. A simple *pre-flow* graph corresponding to simple constraints (closely resembling the syntax tree for the analysed term) is *closed* according to a set of rules corresponding to the conditional constraints in Palsberg's analysis. Solving constraints in Palsberg's formulation corresponds to *reachability* in graphs.

Section 4.2 presents a different form of graph where standard type information is represented explicitly in the graph. A *typed* flow graph will generally be bigger than an untyped one, but avoids the need for the closing rules: a graph containing the full flow information is generated directly from the standard type inference tree. Section 4.3 proves that typed and untyped graphs compute the same flow information. (A direct proof of soundness can be found in [Mos97b]).

The graph formulation does not lead to any algorithmic improvement over previous formulations. It does, however, serve to identify the complexity: in particular, if the size of all types in the program is bounded by a constant, full flow analysis can be done in *quadratic* time. Furthermore, this formulation allows *demand driven* analysis: a specific questions (such as "which functions can be applied at $@^l$") can be answered without computing the full result of the analysis. Under assumption that types are bounded, such queries can be answered in *linear* time. Modularity and algorithms are considered in section 4.4.

In section 4.5 we will see that the paths defined in this chapter are equiv-

alent to the notion of *well-balanced path* by Asperti and Laneve [AL93]. Asperti and Laneve refine the concept of well-balanced path to *legal path* which captures exact set of virtual redexes in a term — in section 8.6 we discuss whether this can be useful as a basis for defining more precise analyses.

Finally, section 4.6 summarises the results obtained in part I.

4.1 Untyped graphs

A flow graph is a structure capturing the possible flow of an expression. A path in a flow graph will represent the flow of a value.

An untyped flow graph is a directed graph (V, E) where V is a set of *vertices* (or *nodes*) and E is a set of *directed edges* (or *arrows*) which is a subset of $V \times V$. We use e to denote edges and n to denote nodes. The set of nodes V contains

1. *Variable* nodes. Exactly one node for each variable (free or bound) in the analysed expression.

2. *Constructor* nodes $\rightarrow^+$, $\times^+$ and Bool^+ which construct function-, pair- and boolean values. They correspond to abstractions, pairs and booleans.

3. *Destructor* nodes $\rightarrow^-$, $\times^-$ and Bool^- that use function-, pair- and boolean values. They correspond to applications, let (x, y) be ... in ... and conditionals.

4. Anonymous box-nodes that represent the *result* of a subexpression.

In addition, we will have nodes 'let' and 'fix': these will *not* be connected to the rest of the graph, and are only included to aid readability.

In a flow graph for expression e each variable occurring in e will be represented by one node whereas each occurrence of a variable will be represented by a box-node. Constructor and destructor subexpressions of e are represented by a constructor or destructor node and a box-node. Let- and fix-expressions are represented by a box node, but we add let- and fix-nodes for readability. Thus for every subexpression e' of e there is one box node which represents the result of e'. The box-node is referred to as the *root* of the graph for e'.

A *path* is defined in the standard fashion. We use p to denote paths.

1. Any arrow is a path

2. If p is a path from n_1 to n_2 and p' is a path from n_2 to n_3 then the composition of p and p' is a path from n_1 to n_3. The composition is written as $p \cdot p'$.

Graphically we use doubly pointed arrows to denote paths:

In a flow graph, value flow is represented by paths: a path from a constructor node to a destructor node represents a potential use of the value constructed by the constructor. The path will traverse variable nodes that can potentially be bound to the value.

4.1.1 Pre-flow-graphs

A *pre-flow-graph* for an expression e is a graph (V, E) which we will use as the basis for constructing the flow graph for e.

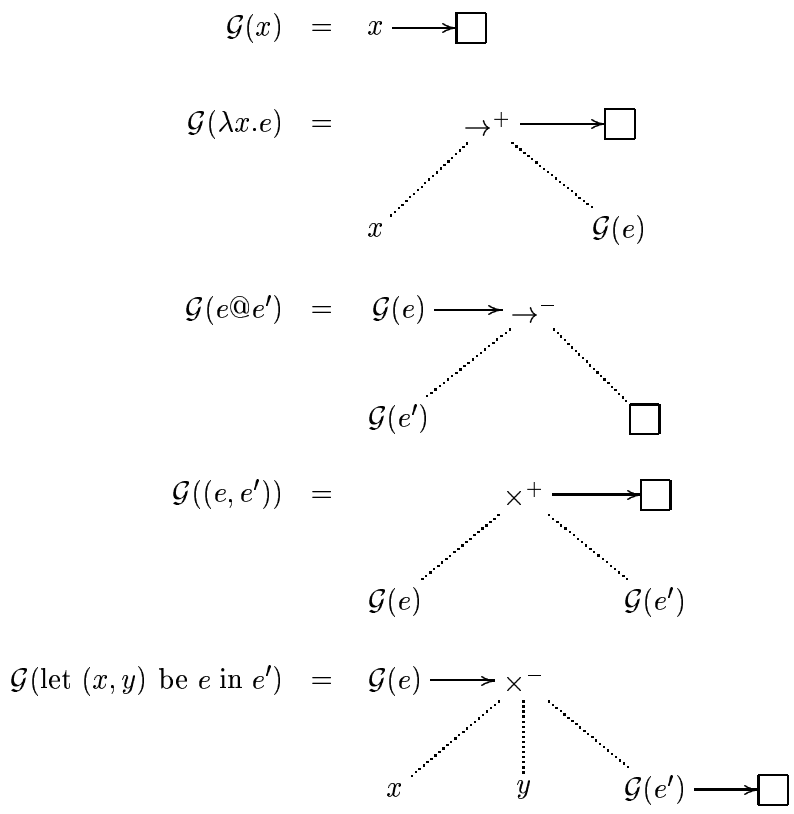


Figure 4.1: Pre-flow-graphs (1)

Figures 4.1 and 4.2 define a function $\mathcal{G}$ mapping (untyped) expressions to pre-flow-graphs. $\mathcal{G}(e)$ is defined inductively over the structure of e. A pre-flow-graph contains only the most rudimentary flow, namely

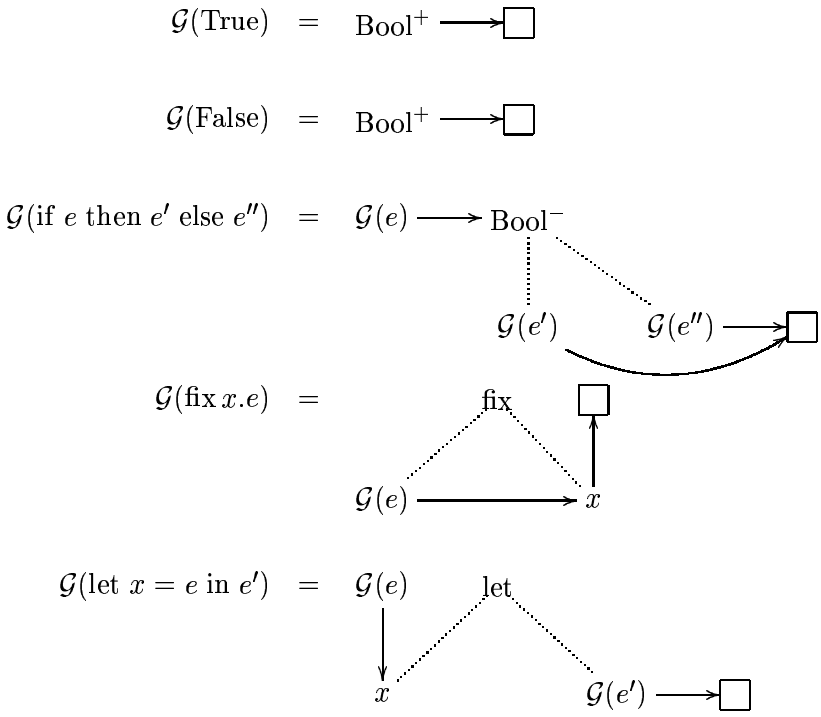


Figure 4.2: Pre-flow-graphs (2)

1. the flow from a constructor to its root,
2. the flow from the root of subgraphs to destructors,
3. the flow from variable to variable occurrence,
4. the flow from the branches of a conditional to the root of the conditional and
5. the flow from let- and fix-bound expressions to the let- resp. fix-bound variables.

The edges in pre-flow-graphs correspond directly to the unconditional constraints generated in the constraint based flow analysis of Palsberg [Pal94].

In the definition we use dotted lines to make the syntax tree for e explicit. These lines are not part of the flow graph, but represent the syntax tree. They are convenient during the construction of the flow graph (in particular the closing rules of the following subsection). The direction and ordering of these dotted lines is important, e.g. the left sibling of a $\rightarrow^-$ node is the argument to the application and the right sibling is the result.

Formally we can use functions *varof* and *bodyof* to give us the variable resp. body node of $\rightarrow^+$ and *argof* and *resultof* to give us the argument resp. result node of $\rightarrow^-$. For pairs we use *fstof* and *sndof* for the components of $\times^+$ and *fstvarof* and *sndvarof* for the variables bound by $\times^-$.

Each case of the definition defines a graph with a root node $\square$. Whenever a right hand-side refers to $\mathcal{G}(e)$, the root node of $\mathcal{G}(e)$ is connected at this place. The root node represents the result of the graph.

For each variable x in expression e (free or bound) there is exactly one node x. Thus the case for variable occurrences in the definition of $\mathcal{G}$ connects this one node to a new box node.

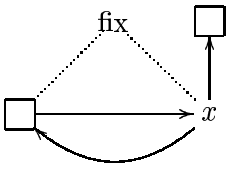
The graph for lambda creates a new node for the constructor called $\rightarrow^+$ and the node for the bound variable. The result of a lambda expression is the lambda, thus the constructor is connected to the root node. In the application case the root of the applied function is connected to the destructor node $\rightarrow^-$ indicating that this is the value that is consumed. The root is unconnected (but will be connected according to what is applied — see subsection 4.1.2).

Pairs are treated like abstractions creating a new $\times^+$ constructor node. Pair destruction is similar to application except that it is a binding construct for two variables and that the body of the let is connected to the root.

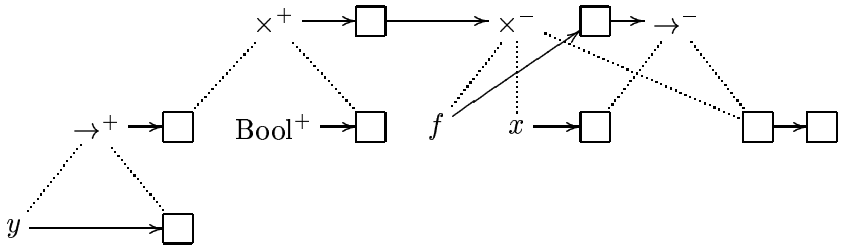
In the graphs we abstract from whether a boolean constructor is True or False; both are mapped to Bool^+. Similarly, in the graph for 'if', we connect both branches to the root node. (It would be more in the spirit of the other rules if we did not connect the branches to the root, and had a closing rule that did this if there was a path from a boolean constructor to the boolean

destructor. This would, however, not correspond to the analyses presented in the previous chapter.[1])

In the fix-case the body is connected to the bound variable reflecting that unfolding will bind the variable to the body. We connect the variable to the root[2]. Note that fix-expressions can make graphs cyclic: e will usually contain x as a free variable. E.g. $\mathcal{G}(\text{fix}\, x.x)$:



Example 4.1 Applying $\mathcal{G}$ to let (f, x) be $(\lambda y.y, \text{True})$ in $f@x$ results in the following pre-flow-graph:



where the rightmost box is the root of the graph.

□

4.1.2 Closing Pre-flow-graphs

Note how a pre-flow-graph contains very limited flow information. The graph $\mathcal{G}((\lambda^{l_1} x.e)@^{l_2} e')$ will contain a path from the node associated with λ^{l_1} to the node associated with $@^{l_2}$ but there are no paths leading from e' to x (and further into e)[3]. This is the purpose of the *closing rules* presented in figure 4.3.

The first rule in figure 4.3 reflects the fact that whenever a function can be applied at some application, the argument can flow into the bound

[1] One could even take this a step further and consider connecting the body of 'let' and pair-destructors to the root as well as connecting the body of 'fix' to the fix-bound variable as closing rules.

[2] We could have connected the body to the root instead. In typed graphs, to be presented later, this would violate an invariant that only variable nodes have more than one exiting arrow.

[3] From this point we will be sloppy and identify constructor/destructor occurrences in expressions with the constructor/destructor nodes associated with them. Similarly, we will talk about the labels of nodes when referring to the label of the associated expression.

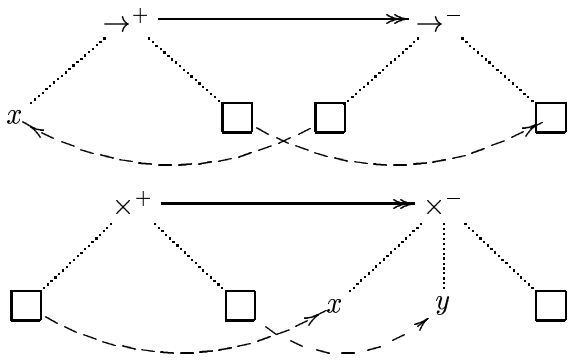


Figure 4.3: Closing rules

variable and the result of the function body can flow to the result of the application. The dashed arrows represent the added edges (and are no different than other edges).

The second rule reflects that if a pair can flow to a pair destructor then the components of the pair can flow to the variables bound by the destructor. The two closing rules correspond directly to the conditional constraints of Palsberg's constraint based analysis [Pal94] (figure 3.5).

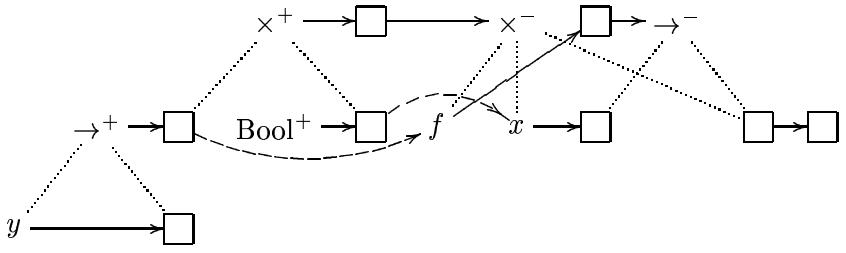
Function '*close*' mapping flow graphs to flow graphs is defined to be the transitive closure of the closing operations defined in figure 4.3.

Definition 4.2 *If G is a flow graph (V, E) then $close(G)$ is the least flow graph (V, E') such that*

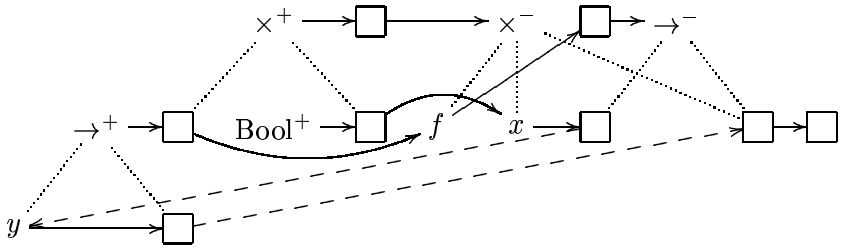
1. *If there is a path from a node $n_1 =\rightarrow^+$ to $n_2 =\rightarrow^-$ then there is an arrow from $argof(n_2)$ to $varof(n_1)$ and an arrow from $bodyof(n_1)$ to $resultof(n_2)$.*

2. *If there is a path from a node $n_1 = \times^+$ to $n_2 = \times^-$ then there is an arrow from $fstof(n_1)$ to $fstvarof(n_2)$ and an arrow from $sndof(n_1)$ to $sndvarof(n_2)$.*

Example 4.3 We show how the pre-flow-graph of example 4.1 is closed. First we notice that there is a path (consisting of two edges) from $\times^+$ to $\times^-$, so we can apply the closing rule as follows (the new edges are dashed):



In this graph, we have created a path from $\rightarrow^+$ to $\rightarrow^-$ going through f so we can apply the closing rule again to obtain:



We now see a path from Bool^+ to the root, reflecting that True will indeed be the result of evaluating the expression.

□

4.1.3 Equivalence to Constraint Based Analysis

We will sketch how the graph based analysis corresponds to Palsberg constraint based analysis [Pal94]. Let an expression e be given. Let ϕ be a map from nodes in a flow graph $\mathcal{G}(e)$ to Palsberg's $X_e \cup Y_e$ defined by:

1. If n is the root of $\mathcal{G}(e')$ then $\phi(n) = [\![e']\!]$.
2. If n is variable x then $\phi(n) = x$.
3. If n is a constructor with label l then $\phi(n) = \{l\}$.
4. If n is a destructor destructing e' then $\phi(n) = [\![e']\!]$.

Eg. ϕ identifies the root of the argument with the node for the application itself.

It is easy to see that if there is an edge from n to n' in $close(\mathcal{G}(e))$ then $\phi(n) \subseteq \phi(n')$ is in the constraint set for e. By transitivity of $\subseteq$ we extend this to paths: if there is a path from n to n' in $close(\mathcal{G}(e))$ then $\phi(n) \subseteq \phi(n')$.

For the other direction we see that if V and V' are in $X_e \cup Y_e$ and $V \subseteq V'$ then for all n, n' such that $\phi(n) = V$ and $\phi(n') = V'$ there exists a path in $close(\mathcal{G}(e))$ from n to n'.

4.2 Typed graphs

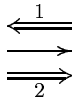
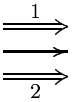
The idea of typed graphs is the same as with untyped graphs: compute a graph for expression e such that the data flow when e is evaluated is represented in the graph as paths. We will, however, in the definition of typed graphs take advantage of the fact that a standard typing of e is given.

A *typed flow graph* for e is a graph (V, E) as above. The principal difference is that a subexpression $e' : t$ is represented by a set of nodes: one

node for each constructor (Bool, $\times$, $\rightarrow$) in t. The node associated with the top type constructor of t is named according to e while the rest are anonymous (but still conceptually associated to this named node). Collections of nodes associated with different subexpressions are connected by collections of edges which intuitively carry values of the appropriate type.

Using the above definition would be cumbersome, so we introduce shorthand graphical notation for typed graphs. We hope that it will be clear that this is indeed nothing but convenient notation.

We represent the set of nodes associated with a subexpression by one (multi-) node — it can be convenient to think of such a multinode as a parallel "plug". The set of edges between two nodes form a *cable*. To be precise, we define a t-cable as follows:

1. A Bool-cable is a single edge (wire): 
2. A $(t \rightarrow t')$-cable is  where  is a t-cable,  is its flipped version and  is a t'-cable.
3. A $(t \times t')$-cable is  where  is a t-cable and  is a t'-cable.

By "flipped" we mean inverting the direction of all wires in the cable but not changing the top to bottom order of wires.

The composition of cables is called a *cable-path* and is written . If c is one of the following cables

the edge e is called the *carrier* of c.

If e is an edge in a cable c, we say that it is a *forward* edge if it has the same direction as the carrier of c, and a *backward* edge if it is in the opposite direction.

Figures 4.4 and 4.5 define a function $\mathcal{TG}$ from expressions to typed flow graphs. As in the definition of $\mathcal{G}$ each right-hand side of the definition has a root (multi-)node which is the node to be connected at recursive calls. Note that each constructor node generates a new carrier starting at the node and connects the sub-cables, while a destructor node terminates a carrier (and connects sub-cables). Note that whenever two cables are connected, they have the same type.

The graphs resulting from $\mathcal{TG}$ is drawn to resemble untyped graphs as much as possible to make comparison easier.

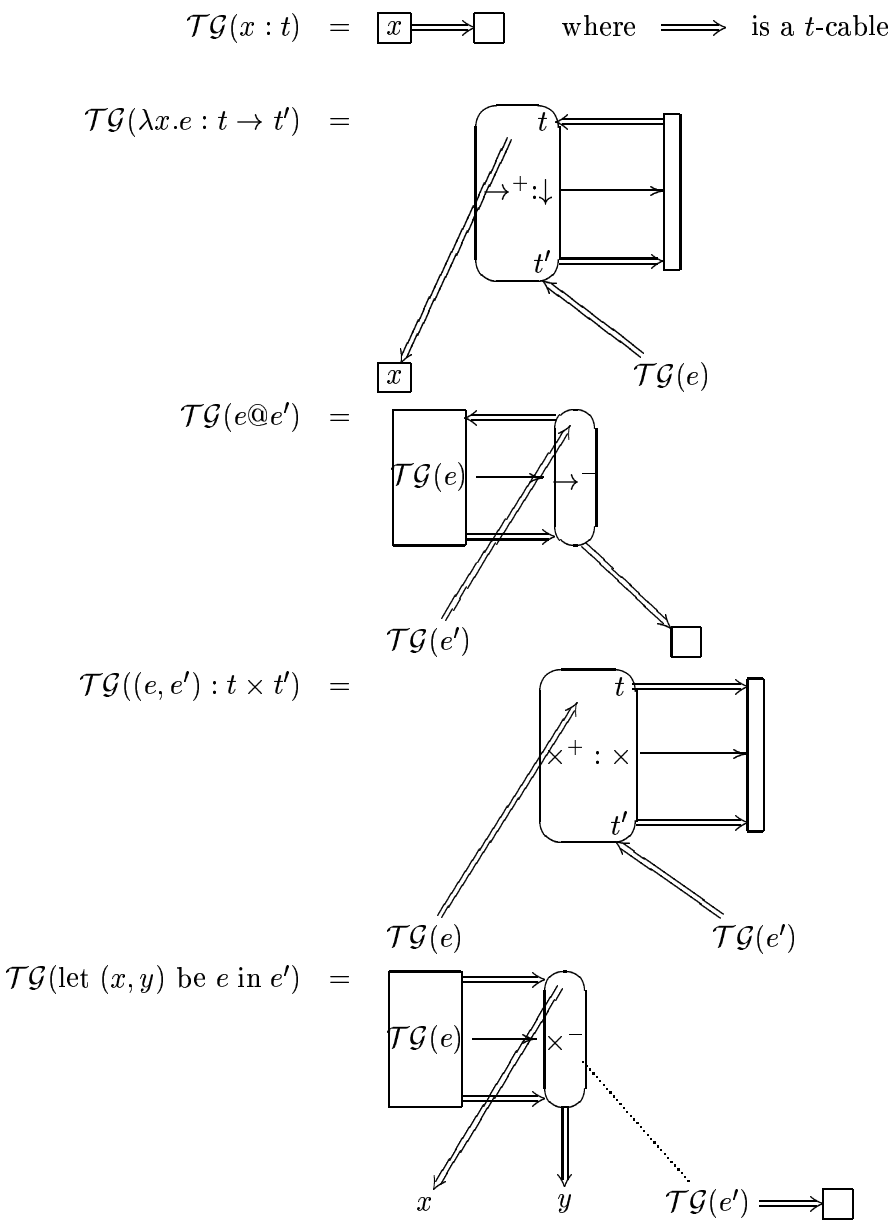


Figure 4.4: Typed flow graphs (1)

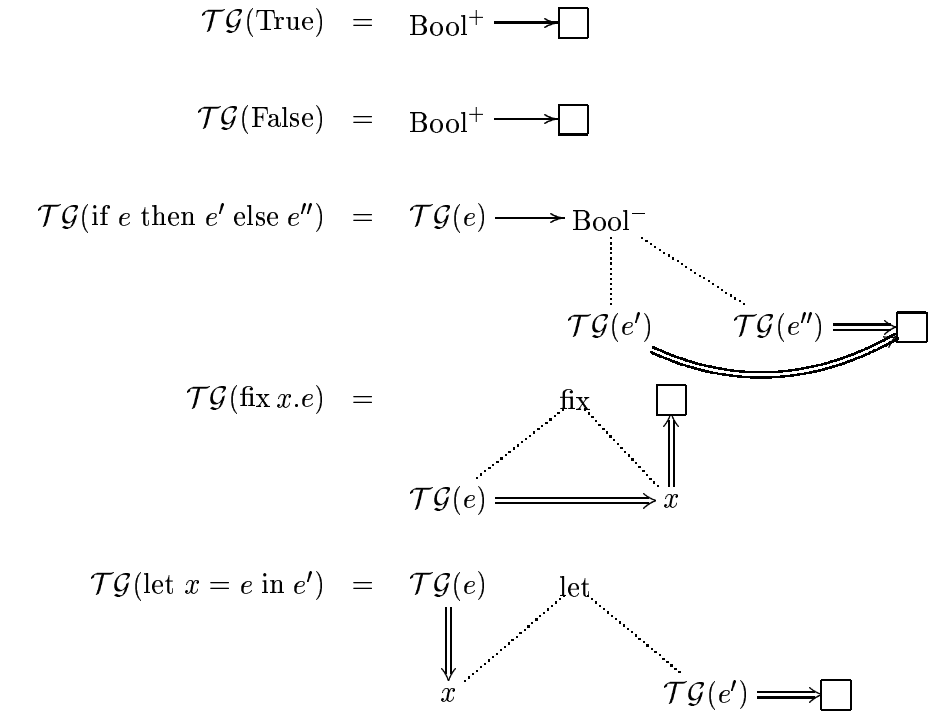


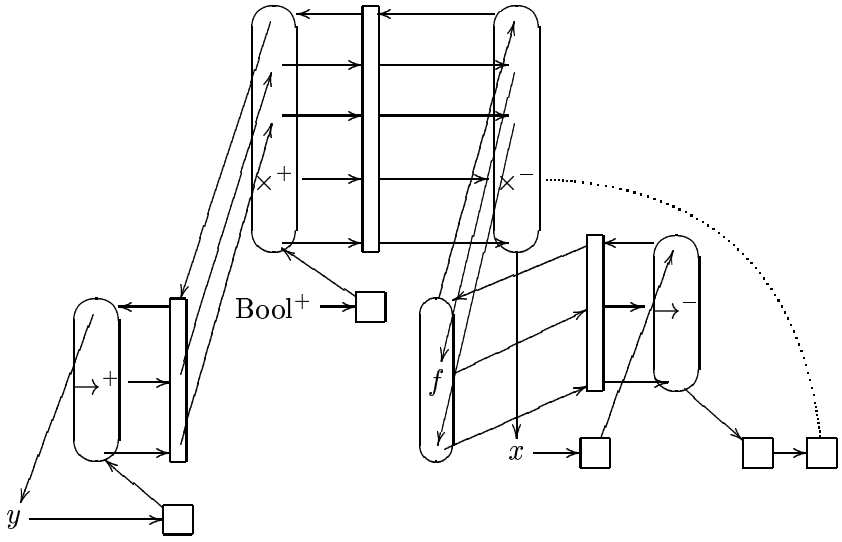
Figure 4.5: Typed flow graphs (2)

In $\mathcal{TG}(\lambda x.e)$, the three cables associated with the $\rightarrow^+$ (multi-)node are called the *binding*-cable (of x), the *body*-cable and the *result*-cable. The downward arrow in the $\rightarrow^+$ node is intended to represent the arrow constructor of the type (which intuitively is carried by the middle edge).

The cables associated with $\rightarrow^-$ (multi-)node are called the *function*-cable, the *argument*-cable and the result-cable. The single cable entering a variable node is called the binding-cable of the variable. For all other (multi-)nodes, we will only need to refer to the cable from the node to the root, which we will call the result-cable.

We show the typed graph obtained for the expression of example 4.1:

Example 4.4 Applying $\mathcal{TG}$ to let (f, x) be $(\lambda y.y, \text{True})$ in $f@x$ results in the following typed flow graph:



The reader is encouraged to follow the path from the Bool$^+$ node (True) to the root of the graph.

□

4.3 Typed and Untyped Graphs

We will consider typed and untyped graphs equivalent if they represent the same flow information. In other words, there should be a path from a constructor to a destructor in the typed graph if and only if there is one in the untyped graph. We use $?^+$-$?^-$ *path* to denote such paths where ? is a type constructor.

For each expression e, there is a one-to-one correspondence between nodes in $\mathcal{G}(e)$ and multi-nodes in $\mathcal{TG}(e)$. When there is no risk of confusion, we will identify these nodes.

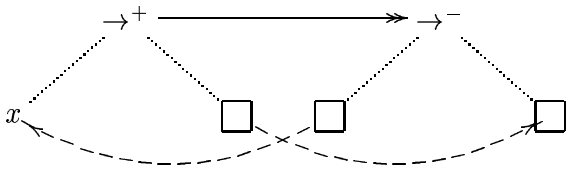
4.3.1 Paths in untyped graphs are in typed graphs

Proposition 4.5 *Let e be a well-typed expression. If p is a path from n_1 to n_2 in $close(\mathcal{G}(e))$ then there exists a path p' from n_1 to n_2 in $\mathcal{TG}(e)$.*

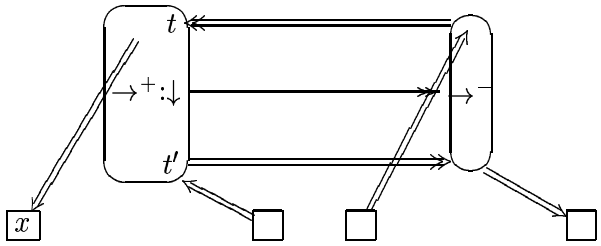
Proof By induction on the computation of $close(\mathcal{G}(e))$

Base: By examination of the definition of $\mathcal{G}$ and $\mathcal{TG}$ we see that all paths in $\mathcal{G}(e)$ are carrier paths in $\mathcal{TG}(e)$.

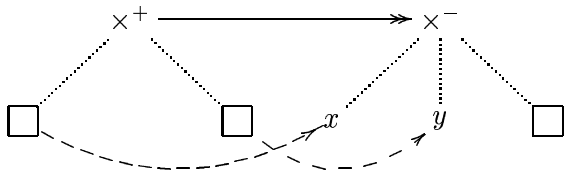
Step: Easy by examining the following pairs of graphs: The new edges of



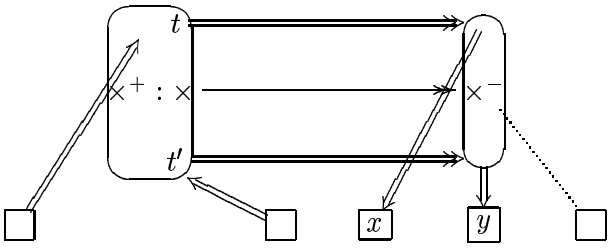
must correspond to paths in



and the new edges in



correspond to paths in



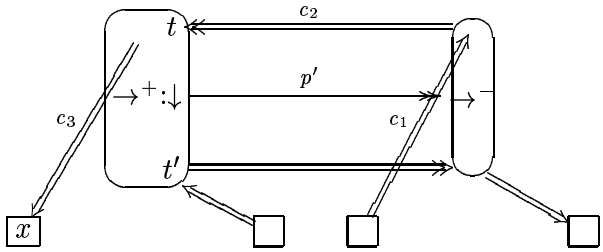
□

4.3.2 Paths in typed graphs are in untyped graphs

Define the *nesting depth* of a carrier edge in a cable c to be 0 and the depth of any other edge e to be 1+ the nesting depth of e in the immediate sub cables of c. A path p is *n-nested* iff the maximal nesting depth of any edge on p is n. We define the concept of *n-level* paths, which we will show is a constructive characterisation of n-nested paths.

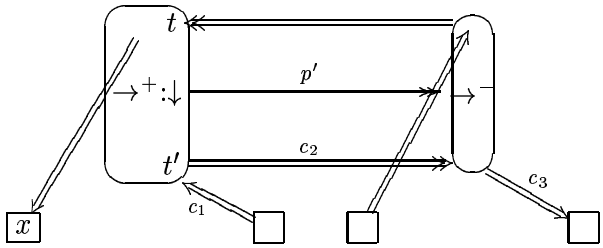
Definition 4.6 *A path in a typed graph is called an* n-level *path iff it is the composition of n-level sub-paths. We say that p is an n-level sub-path iff one of the following holds*

1. *p is the carrier of a cable.*
2. *p is the carrier on the cable-path $c_1 :: c_2 :: c_3$ in a subgraph:*



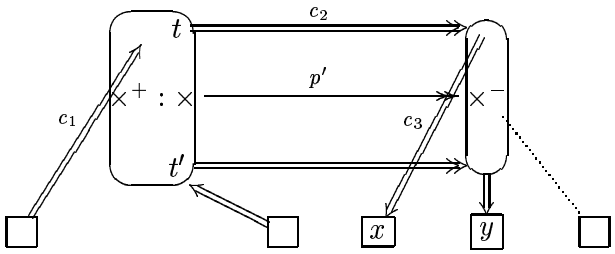
   where p' is a m-level path with $m < n$.

3. *p is the carrier on the cable-path $c_1 :: c_2 :: c_3$ in a subgraph:*



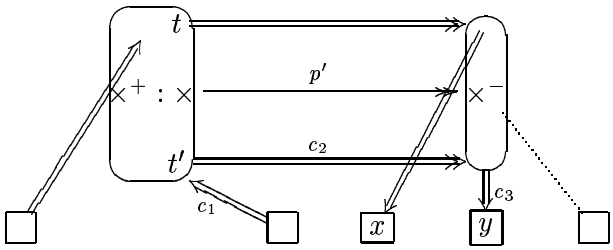
   where p' is a m-level path with $m < n$.

4. *p is the carrier on the cable-path $c_1 :: c_2 :: c_3$ in a subgraph:*



   where p' is a m-level path with $m < n$.

5. *p is the carrier on the cable-path $c_1 :: c_2 :: c_3$ in a subgraph:*



where p' is a m-level path with $m < n$.

Note that all n-level paths are also m-level paths for $m > n$.

Lemma 4.7 *Any n-nested path is an n-level path.*

Proof Let p be an n-nested path. We prove by induction over n that p must be a n-level path:

Base: $n = 0$ We see that p must be a carrier.

Step: $n + 1$ We can divide p into $p_1 \cdot p_1' \cdot p_2 \cdots p_m \cdot p_m' \cdot p_{m+1}$ where all p_i sub-paths are carriers and p_i' are not. For every i, the node between p_i and p_i' must be a constructor node $?_i^+$ and the node between p_i' and p_{i+1} must be a destructor node $?_i^-$: in other words it must be one of the four combinations of definition 4.6. The carrier path between $?_i^+$ and $?_i^-$ must have maximal nesting depth $\leq n$ but then by induction p_i' is a n-level sub-path.

□

Since any path must have a maximal nesting depth we have the following corollary:

Corollary 4.8 *Given any expression e. For all $?^+$-$?^-$ paths p in $\mathcal{TG}(e)$ there exists n such that p is a n-level path.*

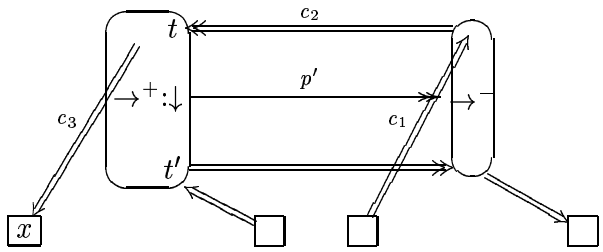
Now we are ready to prove the proposition stating that all interesting paths in typed graphs are also in untyped graphs.

Proposition 4.9 *Let e be any well-typed expression. If p is a $?^+$-$?^-$ path in $\mathcal{TG}(e)$ then there is also a $?^+$-$?^-$ path in $close(\mathcal{G}(e))$.*

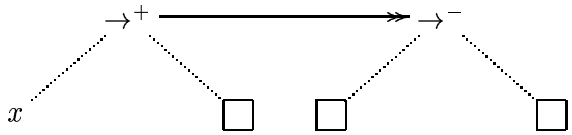
Proof By corollary 4.8 we have that p must be an n-level path for some n. The proof proceeds by induction on n:

$n = 0$: 0-level paths consist only of carrier sub-paths which are all in $\mathcal{G}(e)$.

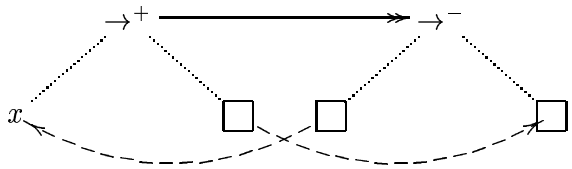
$n > 0$: p can be divided into n-level sub-paths $p_1 :: \cdots :: p_k$. Now, p_i is either a 0-level sub-path in which case we are done trivially or it has one of the forms of definition 4.6:

- 

  where p' is a m-level path with $m < n$. By induction there is a $\rightarrow^+$-$\rightarrow^-$ path in $close(\mathcal{G}(e))$. Thus $close(\mathcal{G}(e))$ contains a subgraph



  but since $close(\mathcal{G}(e))$ is closed it must look like:



- The three remaining cases are similar.

□

We have thus proved that all $?^+$-$?^-$ paths in $\mathcal{TG}(e)$ are in $close(\mathcal{G}(e))$ and vice versa. The result extends, such that a $?^+$-$?^-$ path traverses a variable node x in $close(\mathcal{G}(e))$ if and only if it traverses the corresponding node in $\mathcal{TG}(e)$ by the carrier.

4.4 Modularity and Algorithms

A flow graph (untyped as well as typed) is a representation of the flow of a program. To extract a flow function from a graph, we compute the *transitive*

closure of the graph. Transitive closure is computable in quadratic time in terms of the number of edges in the graph. *Single sink* transitive closure can be computed in linear time (again in the number of edges) allowing us to pose queries such as "given l, what is $\mathcal{F}(l)$?", i.e. "which values are used at l?". Similarly, *single source* transitive closure can be computed in linear time allowing queries "where can value l flow to?".

Dynamic transitive closure is at the heart of the constraint based algorithm of Palsberg [Pal94] — since the best known algorithm for dynamic transitive closure is $\mathcal{O}(n^3)$, this was believed to be also the best obtainable complexity for closure analysis.

The size of a typed graph is proportional to the size of the underlying expression e with explicit types on all subexpressions. This is in general exponential in the size of the untyped expression. Hence, we must expect exponential worst-case behaviour.

In practice, programmers do not write programs with huge types. If we assume that all types are bounded, the size of an expression with explicit types on all subexpressions is proportional to the underlying untyped term. Therefore the number of edges in $\mathcal{TG}(e)$ is proportional to the size of e.

We can conclude that under the assumption that types are bounded, we can compute the flow function in quadratic time and answer flow queries in linear time.

Computing flow information using untyped flow graphs cannot be done modularly. The context of an expression e can easily trigger closing rules — the added edges can even trigger closing rules within $\mathcal{G}(e)$.

Building a typed flow graph proceeds in an entirely modular manner so we can compute the flow graph separately for different modules.

The flow function computed by transitive closure over a graph is not immediately modular, but can easily be made so: consider the multi-node for a free variable $x : t$ as a constructor node with special unique labels α_i on every type constructor in t occurring in positive position (e.g. if x : Bool $\to$ Bool is free in e a fresh label $\alpha_\to$ can enter $\mathcal{TG}(e)$ along the carrier to every occurrence of x and a fresh label α_{Bool} can enter along the result edge). Similar labels are added to negative edges in the result cable. Using transitive closure now computes a polymorphic flow function that can be instantiated by the context.

4.5 Paths by Asperti and Laneve

Paths play a crucial role in the field of optimal reduction. Levy defined a notion of two redexes being created in the "same" way during reduction in which case they were said to belong to the same "family" [Lév78, Lév80]. Reduction was said to be optimal if a family of redexes was only reduced once.

Later Asperti and Laneve [ADLR94] identified families of redexes with *legal* paths — a legal path identifies a virtual redex (family of redexes) that in order to do optimal reduction may not be copied. In the field of optimal reduction it was realised for legal paths that:

> "Intuitively, this "path" describes the "control flow" between the application and the associated $\lambda\ldots$"
>
> [AG96] page 107

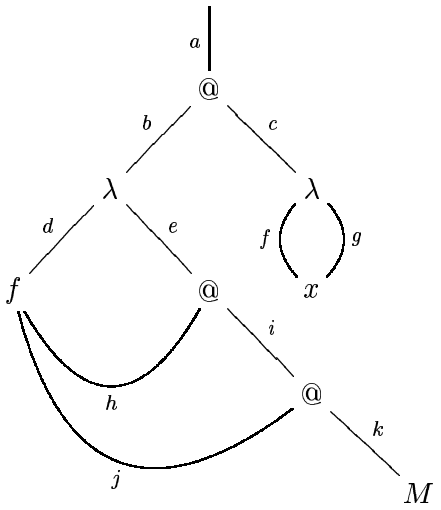
To my knowledge this message has not gotten through to the program analysis community even though in a certain sense an *exact* analysis is given.

In this section, we will not give the definition of legal paths (this will be given in section 8.6), only give the definition of well-balanced path which is a superset of legal paths and which we will see is equivalent to paths in our untyped (and thus also typed) graphs.

Well-balanced paths are defined over syntax trees (seen as undirected graphs) with the following extra properties:

- Variable occurrences are connected back to the unique lambda-bound variable.
- The (undirected) edges are given a unique name (for reference).

Example 4.10 The term $(\lambda f.f@(f@M))@(\lambda x.x)$ is represented by:

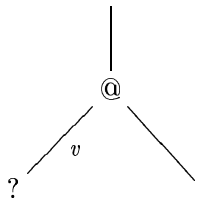


□

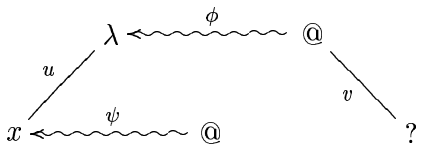
We follow [AG96] closely but not exactly — in particular we make variables an explicit part of the graph, while Asperti and Guerrini identify variables and their binders (i.e. have nodes λx and occurrences of x represented by an edge to this node). The difference is immaterial. If ϕ is a path, reversing ϕ is denoted by $(\phi)^r$. Well-balanced paths are defined as follows:

Definition 4.11 *A* well-balanced *path (*wbp*) of type* @-? *(where* ? *is one of var (variable),* λ *or* @*) is defined inductively as follows*

1. *A function edge v from an application to a node* ? *is a well-balanced path of type* @$-$?.

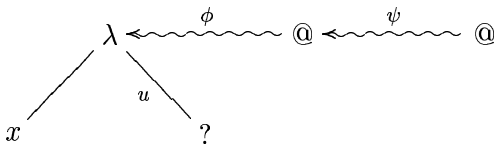


2. *Let ψ be a wbp of type* @-*var, u be the edge from the variable to its binding lambda, ϕ a wbp of type* @-λ *ending at this lambda and v be the argument edge of the initial node of ϕ.*



Then $\psi u(\phi)^r v$ is a wbp of type @-? *where* ? *is the node connected to v.*

3. *Let ψ be a wbp of type* @-@ *ending in some node n, ϕ be a wbp of type* @-λ *starting at n and let u be the body-edge of the lambda.*



Then $\psi\phi u$ is a wbp of type @-? *where* ? *is the node connected to u.*

In example 4.10 above b, h and j form simple wbp's and *hdbc* and *jdbc* forms wbp's.

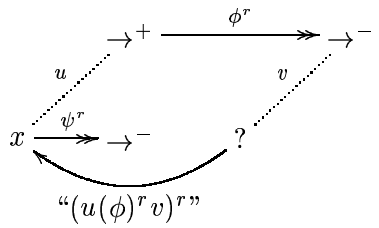
We will show that the paths in the untyped and typed graphs (restricted to the lambda-calculus) are exactly Asperti's well-balanced paths: there is a wbp of type @-? in Asperti's syntax-trees if and only if there is a path leaving the root node corresponding to ? leading to the node corresponding to the @-node in the untyped/typed graph for the same expression. Remember, that when ? is an application there is no path from the application node ($\rightarrow^-$) itself to the root node.

We will see that case *1.* corresponds to the construction of pre-flow-graphs and cases *2.* and *3.* correspond to the $\rightarrow^+$-$\rightarrow^-$ closing rule. We

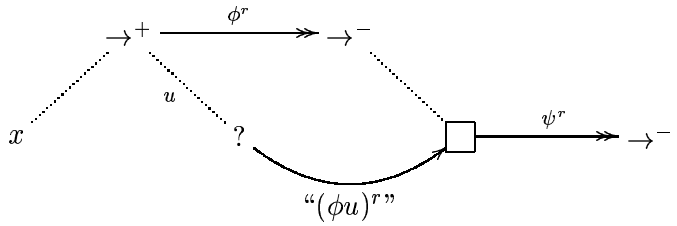
first show that any wbp in the syntax tree for e corresponds to a path in $close(\mathcal{G}(e))$.

Let e be any lambda-expression. Consider *1.* in the inductive definition of wbp's: a path from @ to ? is a wbp according to this base case if and only if there is a path from the root of ? to @ in the pre-flow-graph $\mathcal{G}(e)$.

Consider the inductive case *2.*: we can assume that there is a path in $close(\mathcal{G}(e))$ corresponding to $(\phi)^r$ and a path corresponding to $(\psi)^r$. By the closing rules there must be a path corresponding to $(u(\phi)^r v)^r$ from the root of ? to x and thus by composing this with the path corresponding to $(\psi)^r$ we find that there is a path corresponding to $(u(\phi)^r v)^r(\psi)^r$ in $close(\mathcal{G}(e))$. The following graph is useful for illustration:



Similarly for *3.*: we have a path (corresponding to $(\psi)^r$) in $close(\mathcal{G}(e))$ from the root of an application to another application and a path (corresponding to $(\phi)^r$) from a lambda to the first application. By the closing rules, the latter path triggers an edge from root of the body of the lambda to the root of the application, which composed with the path corresponding to $(\psi)^r$ proves that there is a path corresponding to $(\psi\phi u)^r$:



For the other direction, any path ending in an application node $\to^-$ must be either in the pre-flow-graph or have the structure of one of the above graphs. This proves the following theorem:

Theorem 4.12 *For any any lambda-expression e the following are equivalent:*

1. *There is a well-balanced path of type* $@^l - ?$ *in the syntax tree for e.*

2. *There is a path in* $close(\mathcal{G}(e))$ *from the root of ? to* $@^l$.

While it was easiest to prove the correspondence using untyped graphs, the actual wbp's resemble paths in typed graphs more closely: there is a one-to-one correspondence between edges in the syntax tree for e and cables in $\mathcal{TG}(e)$. Then wbp's and paths ending with a carrier edge entering a $\rightarrow^-$ node in $\mathcal{TG}(e)$ will be exactly the reverse of each other under this correspondence.

4.6 Summary of Monovariant Analyses

In part I we have presented a number of monovariant analyses. Simple flow analysis was equivalent to an analysis by Bondorf and Jørgensen[BJ93] and Heintze [Hei95] but allowed modular analysis due to the principal typing property. Analyses based on subtypes, untyped graphs and typed graphs were equivalent to analyses by Sestoft [Ses88, Ses91] and Palsberg [Pal94]. We improved over their analyses by:

- Giving a modular analysis.
- Reducing the complexity to quadratic under assumption the types are bounded.
- Allowing single queries in linear time — posing queries one by one is not asymptotically worse than computing the flow function.

Part II

Polyvariant Analysis

Chapter 5

Polymorphism

This chapter extends the subtype flow analysis of chapter 3 with polymorphism. We will present two versions of polymorphic flow analysis: ML-polymorphic flow analysis (also known as let-polymorphism, described in section 5.2) and flow analysis with polymorphic recursion (also known as fix-polymorphism, described in section 5.3). We will give algorithms for computing principal typings in the two polymorphic systems, both of which are of polynomial complexity.

The analyses presented in this chapter are to our knowledge new. Many of the ideas have been used in the context of polymorphic binding-time analysis in papers by Henglein and the present author [HM94] and by Dussart, Henglein and the present author [DHM95a].

Polymorphism allows definitions to be reused in different contexts without the different uses interfering with each other. Let- and fix-polymorphism allow this for let-bound resp. fix-bound expressions. Thus, if a (possibly recursively) defined variable is used in multiple contexts a fresh instance of the type is allowed in each use. This is strong enough to prove subject expansion for let- and fix-reduction.

5.1 Polymorphic Formulae and Logical Rules

The formulae and subtype relation are common to the ML-polymorphic and fix-polymorphic systems and are presented in figure 5.1.

We have two kinds of formulae in polymorphic flow analysis. *First-order flow formulae* $\mathcal{K}^\forall(t)$ are exactly the same as the flow formulae we have seen for simple and subtype flow analysis. In addition to this we also define for every t the set of *predicative second order flow formulae* or *flow schemes* $\mathcal{S}^\forall(t)$. We let σ range over flow schemes. A flow scheme has the form $\forall\vec{\alpha}.C \Rightarrow \kappa$ where κ is a first-order flow formula and C is a constraint set $\{\ell_1 \subseteq \beta_1, \cdots, \ell_n \subseteq \beta_n\}$ and $\vec{\alpha}$ is a list of flow variables $\langle\alpha_1, \cdots, \alpha_m\rangle$. The constraints in C are called the *qualifiers* of σ and the variables in $\vec{\alpha}$ are

Formulae:

$$\text{Bool}\ \frac{}{\text{Bool}^{\ell} \in \mathcal{K}^{\forall}(\text{Bool})}$$

$$\to\ \frac{\kappa \in \mathcal{K}^{\forall}(t) \quad \kappa' \in \mathcal{K}^{\forall}(t')}{\kappa \to^{\ell} \kappa' \in \mathcal{K}^{\forall}(t \to t')} \qquad \times\ \frac{\kappa \in \mathcal{K}^{\forall}(t) \quad \kappa' \in \mathcal{K}^{\forall}(t')}{\kappa \times^{\ell} \kappa' \in \mathcal{K}^{\forall}(t \times t')}$$

$$\forall\ \frac{\kappa \in \mathcal{K}^{\forall}(t)}{\forall \vec{\alpha}.C \Rightarrow \kappa \in \mathcal{S}^{\forall}(t)}$$

Subtype relation:

$$\text{Bool}\ \frac{C \vdash \ell_1 \subseteq \ell_2}{C \vdash^{\forall} \text{Bool}^{\ell_1} \leq \text{Bool}^{\ell_2}}$$

$$\text{Arrow}\ \frac{C \vdash^{\forall} \kappa_1 \leq \kappa_1' \quad C \vdash^{\forall} \kappa_2 \leq \kappa_2' \quad C \vdash \ell_1 \subseteq \ell_2}{C \vdash^{\forall} \kappa_1' \to^{\ell_1} \kappa_2 \leq \kappa_1 \to^{\ell_2} \kappa_2'}$$

$$\text{Product}\ \frac{C \vdash^{\forall} \kappa_1 \leq \kappa_1' \quad C \vdash^{\forall} \kappa_2 \leq \kappa_2' \quad C \vdash \ell_1 \subseteq \ell_2}{C \vdash^{\forall} \kappa_1 \times^{\ell_1} \kappa_2 \leq \kappa_1' \times^{\ell_2} \kappa_2'}$$

Figure 5.1: Polymorphic flow analysis — formulae and subtype relation

called the *quantifiers*.

The intended reading of $e : \forall\vec{\alpha}.C \Rightarrow \kappa$ is that e has type $\kappa[\vec{\ell}/\vec{\alpha}]$ for all *instantiations* of $\vec{\alpha}$ such that $C[\vec{\ell}/\vec{\alpha}]$ is provable.

The subtype relation of figure 5.1 is identical to the similar relation of subtyping flow analysis (figure 3.1).

5.2 ML polymorphism

ML-polymorphic flow analysis[1] allows flow schemes in let-bound expressions only.

The non-logical rules of figure 5.2 only differ from the subtype flow analysis in one rule: the let-bound expression in the rule for let is allowed to have a flow scheme as property. This allows different instantiations of the flow scheme for each occurrence of the let-bound variable. The qualifications of the flow scheme are constraints that must be true for each of the instantiations.

Furthermore, two new semi-logical rules are added: the ($\forall$-I) rule allows us to *qualify* over any subset of the constraint assumptions and to *quantify* over any vector of flow variables provided none of the variables are free in the (non-qualified) constraint set nor in the environment. The ($\forall$-E) rule allows us to instantiate quantified flow variables provided the qualifying constraints are provable.

We first present a syntax directed version of the polymorphic type system. This allows us to give an algorithm for computing principal types as in the previous chapters, but for polymorphic analyses it will also be the starting point for our definition of flow functions.

In contrast with the subtype system of chapter 3, existence of principal types is not the only aim of giving an algorithm — the algorithm itself is of interest. Since ML-polymorphic flow analysis is new, naturally no algorithms exists and we are obliged to substantiate its raison d'être by a complexity argument.

5.2.1 Syntax Directed Type System

We note that the constraint weakening rule:

$$\text{weak}\ \frac{C; A \vdash^{ML} e : \sigma \quad C' \vdash C}{C'; A \vdash^{ML} e : \sigma}$$

is admissible as in chapters 2 and 3[2]. For the rest of the chapter, it will be convenient to allow this rule in our system.

[1] ML-polymorphism refers to the restriction to polymorphism in let-expressions. ML-polymorphism in standard types is the subject of section 9.1.

[2] Formally, we need a side-condition "no variable bound in σ is free in C'". Since it is easy to prove that if $C; A \vdash^{ML} e : \sigma$ (without the weakening rule) then no variable bound

Non-logical rules:

$$\text{Id}\ \frac{}{C; A, x : \sigma \vdash^{ML} x : \sigma}$$

$$\rightarrow\text{-I}\ \frac{C; A, x : \kappa \vdash^{ML} e : \kappa'}{C; A \vdash^{ML} \lambda^l x.e : \kappa \rightarrow^{\{l\}} \kappa'}$$

$$\rightarrow\text{-E}\ \frac{C; A \vdash^{ML} e : \kappa' \rightarrow^{\ell} \kappa \quad C; A \vdash^{ML} e' : \kappa'}{C; A \vdash^{ML} e@^l e' : \kappa}$$

$$\text{Bool-I}\ \frac{}{C; A \vdash^{ML} \text{True}^l : \text{Bool}^{\{l\}}} \qquad \frac{}{C; A \vdash^{ML} \text{False}^l : \text{Bool}^{\{l\}}}$$

$$\text{Bool-E}\ \frac{C; A \vdash^{ML} e : \text{Bool}^{\ell} \quad C; A \vdash^{ML} e' : \kappa \quad C; A \vdash^{ML} e'' : \kappa}{C; A \vdash^{ML} \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

$$\times\text{-I}\ \frac{C; A \vdash^{ML} e : \kappa \quad C; A \vdash^{ML} e' : \kappa'}{C; A \vdash^{ML} (e, e')^l : \kappa \times^{\{l\}} \kappa'}$$

$$\times\text{-E}\ \frac{C; A \vdash^{ML} e : \kappa \times^{\ell} \kappa' \quad C; A, x : \kappa, y : \kappa' \vdash^{ML} e' : \kappa''}{C; A \vdash^{ML} \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{fix}\ \frac{C; A, x : \kappa \vdash^{ML} e : \kappa}{C; A \vdash^{ML} \text{fix}\, x.e : \kappa} \qquad \text{let}\ \frac{C; A \vdash^{ML} e : \sigma \quad C; A, x : \sigma \vdash^{ML} e' : \kappa'}{C; A \vdash^{ML} \text{let}\ x = e\ \text{in}\ e' : \kappa'}$$

Semi-logical rules:

$$\text{Sub}\ \frac{C; A \vdash^{ML} e : \kappa \quad C \vdash^{\forall} \kappa \leq \kappa'}{C; A \vdash^{ML} e : \kappa'}$$

$$\forall\text{-I}\ \frac{C, C'; A \vdash^{ML} e : \kappa}{C; A \vdash^{ML} e : \forall \vec{\alpha}.C' \Rightarrow \kappa}\ \vec{\alpha}\ \text{not free in}\ C, A$$

$$\forall\text{-E}\ \frac{C; A \vdash^{ML} e : \forall \vec{\alpha}.C' \Rightarrow \kappa \quad C \vdash C'[\vec{\ell}/\vec{\alpha}]}{C; A \vdash^{ML} e : \kappa[\vec{\ell}/\vec{\alpha}]}$$

Figure 5.2: ML-polymorphic flow analysis — non-logical rules

It is easy to see that except for trivial uses, instantiation is only applicable immediately after the variable rule and generalisation only as the last rule in the derivation for let-bound expression. Furthermore, using the whole constraint set as qualification can only lead to more general types. These observations along with building in subtype steps leads us to the syntax directed version of the ML-polymorphic system presented in figure 5.3.

The system contains the same rules as the syntax directed system in figure 3.3 with the following differences:

$$\text{Id}\ \frac{C \vdash C'[\vec{\ell}/\vec{\alpha}]}{C; A, x : \forall\vec{\alpha}.C' \Rightarrow \kappa \vdash_n^{ML} x : \kappa[\vec{\ell}/\vec{\alpha}]}$$

$$\text{let}\ \frac{C; A \vdash_n^{ML} e : \kappa \quad C'; A, x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{ML} e' : \kappa'}{C'; A \vdash_n^{ML} \text{let } x = e \text{ in } e' : \kappa'}\ (*)$$

Where $(*)$ means $(\vec{\alpha} \cap (\mathrm{FV}(A) \cup \mathrm{FV}(C'))) = \emptyset$.

Figure 5.3: Syntax directed ML-polymorphic flow analysis

Theorem 5.1 *Type system $\vdash_n^{ML}$ is sound and complete w.r.t. $\vdash^{ML}$:*

Soundness *If $C; A \vdash_n^{ML} e : \kappa$ then $C; A \vdash^{ML} e : \kappa$*

Completeness *If $C; A \vdash^{ML} e : \kappa$ then there exists κ' such that $C; A \vdash_n^{ML} e : \kappa'$ and $C \vdash^{\forall} \kappa' \leq \kappa$.*

Proof The proof of soundness proceeds by induction over the derivation of $C; A \vdash_n^{ML} e : \kappa$

The proof of completeness proceeds by induction over the derivation of $C; A \vdash^{ML} e : \kappa$ (note that we need the constraint weakening rule).

□

5.2.2 What is the Result of Polymorphic Flow Analysis

As in the previously defined analyses, we can regard derivations as the result of the analysis — we will sketch how a ground flow function can be extracted from the derivation if needed. The discussion of this subsection applies also when we extend the system with polymorphic recursion in section 5.3.

Extracting a ground flow function is not as simple as in the monovariant cases since merely extracting the annotation of the top constructor of the

---

by σ is free in C, A, we can always choose a constraint-set C'' alpha-equivalent to C' such that $C'' \vdash C'$ and $C' \vdash C''$

type of an expression will lead us to conclude that the flow of an expression e can be α where α is locally bound. In this case there must exist an expression let $x = e_1$ in e_2 where e is a subexpression of e_1 and x has type $\forall\vec{\alpha}.C \Rightarrow \kappa$ where α is amongst $\vec{\alpha}$. For each occurrence $x^{(i)}$ of x in e_2, there will be an associated use of the instantiation rule which will substitute some ℓ_i for α. The flow of e is then $\bigsqcup_i \ell_i$. This will also instantiate qualifications, retaining provability. Note that this process might have to be repeated since ℓ_j might be another bound variable for some j.

Example 5.2 Consider:

$$\begin{array}{l}\text{let } id = \lambda^{l_1}x.\text{if}^{l_5}\ x \text{ then } x \text{ else } x\\ \text{in } (id@\text{True}^{l_2}, id@\text{False}^{l_3})^{l_4}\end{array}$$

The type of $\lambda^{l_1}x.\text{if}^{l_5}\ x$ then x else x could be $\forall\alpha.\text{Bool}^{\alpha} \rightarrow^{\{l_1\}} \text{Bool}^{\alpha}$ and thus the flow of x is α. To find the ground flow of $\mathcal{F}(l_5)$ we find all uses of id (since this was the 'let' corresponding to the binding of α) and realise that the two applications of the instantiation rule instantiate α to $\{l_2\}$ resp. $\{l_3\}$. We can conclude that the flow is $\mathcal{F}(l_5) = \{l_2, l_3\}$.

Note that instantiating α in the let-bound expression does not affect the results of applying id. The description of the resulting pair will remain $\text{Bool}^{\{l_2\}} \times \text{Bool}^{\{l_3\}}$. □

This can be viewed as exploiting polymorphism *during* analysis and *then* reducing the abstractions and instantiations over labels. That is, in each use of a polymorphic expression, the flow information of other uses does not degrade the result, but within the polymorphic expression, all uses flow together. In the example, the result of the expression is the pair of $\{l_2\}$ and $\{l_3\}$ despite the fact that the flow of x is $\{l_2, l_3\}$.

Note, that there is no derivation that corresponds to this result: if we insist that the type of x in the example is $\text{Bool}^{\{l_2,l_3\}}$ polymorphic flow analysis cannot infer the precise description of the resulting pair.

We will call this way of interpreting the result *sticky* since the annotations of all values that can pass through a variable "stick" to the variable. We will now give a formal definition of a (sticky) ML-polymorphic flow function. Let a flow environment ϕ map pairs of program variables and labels to flow properties. In figure 5.4 we define a function $\Phi_{\mathcal{L}}$ which maps a derivation, a label and a flow environment to a property. If $\mathcal{T} = \dfrac{\mathcal{T}'}{C; A \vdash_n^{ML} e : \kappa}$, we define the flow function for polymorphic flow analysis by

$$\mathcal{F}_{\mathcal{T}}(l) = \Phi_{\mathcal{L}}(\mathcal{T}, l, \phi_0)$$

where

$$\begin{array}{l}\phi_0(x, l) = \{\} \quad \text{for all } x \in FV(e) \text{ and } l \in \mathcal{L} \text{ and}\\ \mathcal{L} = \mathcal{L}_e\end{array}$$

$\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi) = \text{case } \mathcal{T} \text{ of}$

$$\text{Id}\frac{}{C;A,x{:}\forall\vec{\alpha}.C \Rightarrow \kappa \vdash^{ML}_n x{:}\kappa[\vec{\ell}/\vec{\alpha}]} \quad : \quad [\vec{\ell}/\vec{\alpha}](\phi(x,l))$$

$$\to\text{-I}\frac{\mathcal{T}'}{C;A \vdash^{ML}_n \lambda^{l'} x.e{:}\kappa \to^{\{l'\}} \kappa'} \quad : \quad \Phi_{\mathcal{L}}(\mathcal{T}', l, \phi')$$

where $\phi' = \phi[(x,l'') \mapsto \{\}]$ for all $l'' \in \mathcal{L}$

$$\to\text{-E}\frac{\dfrac{\mathcal{T}'}{C;A \vdash^{ML}_n e{:}\kappa' \to^{\ell} \kappa} \quad \mathcal{T}''}{C;A \vdash^{ML}_n e @^{l'} e'{:}\kappa}$$

$$: \quad \Phi_{\mathcal{L}}(\frac{\mathcal{T}'}{C;A \vdash^{ML}_n e{:}\kappa' \to^{\ell} \kappa}, l, \phi) \sqcup \Phi_{\mathcal{L}}(\mathcal{T}'', l, \phi) \sqcup \ell \qquad \text{if } l = l'$$

$$: \quad \Phi_{\mathcal{L}}(\frac{\mathcal{T}'}{C;A \vdash^{ML}_n e{:}\kappa' \to^{\ell} \kappa}, l, \phi) \sqcup \Phi_{\mathcal{L}}(\mathcal{T}'', l, \phi) \qquad \text{otherwise}$$

$$\text{Bool-I}\frac{}{C;A \vdash^{ML}_n \text{True}^{l}{:}\text{Bool}^{\{l\}}} \quad : \quad \{\}$$

$$\text{Bool-I}\frac{}{C;A \vdash^{ML}_n \text{False}^{l}{:}\text{Bool}^{\{l\}}} \quad : \quad \{\}$$

$$\text{Bool-E}\frac{\dfrac{\mathcal{T}_1}{C;A \vdash^{ML}_n e{:}\text{Bool}^{\ell}} \quad \mathcal{T}_2 \quad \mathcal{T}_3}{C;A \vdash^{ML}_n \text{if}^{l'}\ e \text{ then } e' \text{ else } e''{:}\kappa}$$

$$: \quad \Phi_{\mathcal{L}}(\frac{\mathcal{T}_1}{C;A \vdash^{ML}_n e{:}\text{Bool}^{\ell}}, l, \phi) \sqcup \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi) \sqcup \Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi) \sqcup \ell \qquad \text{if } l = l'$$

$$: \quad \Phi_{\mathcal{L}}(\frac{\mathcal{T}_1}{C;A \vdash^{ML}_n e{:}\text{Bool}^{\ell}}, l, \phi) \sqcup \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi) \sqcup \Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi) \qquad \text{otherwise}$$

$$\times\text{-I}\frac{\mathcal{T}' \quad \mathcal{T}''}{C;A \vdash^{ML}_n (e,e')^{l}{:}\kappa \times^{\{l\}} \kappa'} \quad : \quad \Phi_{\mathcal{L}}(\mathcal{T}', l, \phi) \sqcup \Phi_{\mathcal{L}}(\mathcal{T}'', l, \phi)$$

$$\times\text{-E}\frac{\dfrac{\mathcal{T}'}{C;A \vdash^{ML}_n e{:}\kappa \times^{\ell} \kappa'} \quad \mathcal{T}''}{C;A \vdash^{ML}_n \text{let}^{l}\ (x,y) \text{ be } e \text{ in } e'{:}\kappa''}$$

$$: \quad \Phi_{\mathcal{L}}(\frac{\mathcal{T}'}{C;A \vdash^{ML}_n e{:}\kappa \times^{\ell} \kappa'}, l, \phi) \sqcup \Phi_{\mathcal{L}}(\mathcal{T}'', l, \phi') \sqcup \ell \qquad \text{if } l = l'$$

$$: \quad \Phi_{\mathcal{L}}(\frac{\mathcal{T}'}{C;A \vdash^{ML}_n e{:}\kappa \times^{\ell} \kappa'}, l, \phi) \sqcup \Phi_{\mathcal{L}}(\mathcal{T}'', l, \phi') \qquad \text{otherwise}$$

where $\phi' = \phi[(x,l'') \mapsto \{\}]$ for all $l'' \in \mathcal{L}$

$$\text{fix}\frac{\mathcal{T}'}{C;A \vdash^{ML}_n \text{fix } x.e{:}\kappa} \quad : \quad \Phi_{\mathcal{L}}(\mathcal{T}', l, \phi')$$

where $\phi' = \phi[(x,l'') \mapsto \{\}]$ for all $l'' \in \mathcal{L}$

$$\text{let}\frac{\mathcal{T}_1 \quad \mathcal{T}_2}{C';A \vdash^{ML}_n \text{let } x{=}e \text{ in } e'{:}\kappa'} \quad : \quad \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi')$$

where $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C;A \vdash^{ML}_n e{:}\kappa}$

$\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C';A,x{:}\forall\vec{\alpha}.C \Rightarrow \kappa \vdash^{ML}_n e'{:}\kappa'}$

$\phi' = \phi[(x,l') \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_1, l', \phi)]$ for all $l'' \in \mathcal{L}$

Figure 5.4: ML-polymorphic flow function

The lemmas, propositions and theorems of this chapter which concerns flow functions, will be expressed in terms of $\Phi_{\mathcal{L}}$ rather that $\mathcal{F}$ as this allows induction arguments. Whenever we state "for all ϕ" we always assume that the domain of ϕ includes the free variables of the expression that it will be used in conjunction with.

We define $\mathcal{F}_{\mathcal{T}}$ for a non-syntax directed $\mathcal{T}$ to be $\mathcal{F}_{\mathcal{T}'}$, where $\mathcal{T}'$ is the syntax-directed derivation constructed by the proof of completeness in theorem 5.1.

Some program analyses are inherently non-sticky and if a flow analysis is to be used as a basis for such analyses, it will not be desirable to do the "post-unfolding" of polymorphism. This is the case for polymorphic binding-time analysis [HM94] where the specialiser using the binding-time information can treat explicit quantification and instantiation exactly as lambda abstraction and application. In this case the sticky presentation of the analysis will even make binding-time analysis unsound (see section 10.3.1).

We can conclude that a flow function can be computed from a polymorphic derivation, but that the derivation itself contains additional information that is important or even crucial to some applications.

5.2.3 Halbstark Instance Relation

In the next subsection, we will prove existence of principal types for the ML-polymorphic flow analysis. We will do this by the same method that we employed in previous chapters: give an algorithm $\mathcal{W}$ which computes such principal types. The lazy (or strong) instance relation employed in chapter 3 turns out to be inconvenient to work with (this was to some extent already the case in chapter 3): it is too liberal to be used in the induction hypothesis. On the other hand, we cannot prove principal typings under the (weak) instance relation of chapter 2, so we are looking for something in between.

The instance relation we will use is called *halbstark* and is strictly smaller than the lazy (strong) instance relation (definition 3.3): if a typing is a halbstark instance of another typing then it is also a lazy instance, but not vice versa. Similarly, the weak instance relation (definition 2.4) is strictly smaller than the halbstark instance relation. This extends to principality, where principality under weak instance implies principality under halbstark instance, which in turn implies principality under lazy (strong) instance.

The main difference between the lazy and halbstark instance relations is that halbstark instance does not allow the environment to change as much as in the lazy instance relation — this matches algorithm $\mathcal{W}$ well as this computes typings given an environment and we use the instance relation only for comparing typings with the same environment.

The halbstark instance relation was introduced by Henglein [Hen96]:

Definition 5.3 *A flow judgement* $C'; A' \vdash^{ML} e' : \kappa'$ *is a* halbstark instance *of* $C; A \vdash^{ML} e : \kappa$ *(by* S*) if* $e = e'$ *and*

1. $C' \vdash S(C)$,
2. $A' = S(A)$ *and*
3. $C' \vdash^{\forall} S(\kappa) \leq \kappa'$

If $C; A \vdash^{ML} e : \kappa$ *is also an instance of* $C'; A' \vdash^{ML} e' : \kappa'$ *then the two judgements are equivalent.*

As in previous chapters, we have that derivable judgements are closed under the instance relation. We extend this proposition to also state what happens to the flow computed.

Proposition 5.4 *Assume*

$$\mathcal{T}_1 = \frac{\mathcal{T}_1'}{C; A \vdash_n^{ML} e : \kappa}$$

and that $C'; A' \vdash_n^{ML} e : \kappa'$ *is a halbstark instance of* $C; A \vdash_n^{ML} e : \kappa$ *by* S*. Then there exists* $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C'; A' \vdash_n^{ML} e : \kappa'}$ *and*
2. *Let* $\mathcal{L}$ *be given. Let* ϕ, ϕ' *be two environments mapping free variables of* e *to properties. Then for all* $l \in \mathcal{L}$ *we have that* $C' \vdash^{\forall} S\phi(x,l) \leq \phi'(x,l)$ *for all* $x \in FV(e)$ *implies*

$$C' \vdash S(\Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)) \leq \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi')$$

Proof Induction over $\mathcal{T}_1$. □

Now we define the notion of generic instance of type schemes. For the ML-polymorphic system it will be a convenient concept while for polymorphic recursion as defined in the following section, it will be crucial in the development.

Since we have included the (admissible) constraint weakening rule, we can give a nice and simple definition, which would otherwise be rather cluttered (though it can be done as shown by Henglein [Hen96]).

Definition 5.5 *A type scheme* σ' *is a* generic instance *of* σ *written* $\sigma \sqsubseteq \sigma'$ *iff* $; x : \sigma \vdash^{ML} x : \sigma'$*. Type schemes* σ *and* σ' *are* equivalent *written* $\sigma \cong \sigma'$ *iff* $\sigma \sqsubseteq \sigma' \sqsubseteq \sigma$.

Without the weakening rule, this definition would not allow us to apply the (∀-E) rule except when the instantiation of bound variables makes the qualifications ground or trivial (since $\vdash C'[\vec{\ell}/\vec{\alpha}]$ in the rule can only be provable if $C'[\vec{\ell}/\vec{\alpha}]$ is ground).

Lemma 5.6 *Let C and C' be constraint sets (that is, they contain only constraints of the form $\alpha \subseteq \beta$ and $L \subseteq \alpha$). We have*

$$\forall\vec{\beta}.C' \Rightarrow \kappa' \sqsubseteq \forall\vec{\alpha}.C \Rightarrow \kappa$$

iff there exists a substitution S working on[3] $\vec{\beta}$ such that

1. $C \vdash S(C')$,
2. $C \vdash^{\forall} S(\kappa') \leq \kappa$, *and*
3. *No α_i is free in $\forall\vec{\beta}.C' \Rightarrow \kappa'$.*

Proof

"if" Since S works on $\vec{\beta}$, the following inference is valid:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{;x:\forall\vec{\beta}.C'\Rightarrow\kappa' \vdash^{ML} x:\forall\vec{\beta}.C'\Rightarrow\kappa'}}{C;x:\forall\vec{\beta}.C'\Rightarrow\kappa' \vdash^{ML} x:\forall\vec{\beta}.C'\Rightarrow\kappa'} \quad C\vdash S(C')}{C;x:\forall\vec{\beta}.C'\Rightarrow\kappa' \vdash^{ML} x:S(\kappa')} \qquad C\vdash^{\forall} S(\kappa')\leq\kappa}{\dfrac{C;x:\forall\vec{\beta}.C'\Rightarrow\kappa' \vdash^{ML} x:\kappa}{;x:\forall\vec{\beta}.C'\Rightarrow\kappa' \vdash^{ML} x:\forall\vec{\alpha}.C\Rightarrow\kappa}\,(*)}$$

where $(*)$ means *($\vec{\alpha}$ not free in $\forall\vec{\beta}.C' \Rightarrow \kappa'$)*. The rules applied are from top to bottom: (weak), (∀-E), (Sub) and (∀-I).

"only if" It is not hard to see that any derivation of $;x:\forall\vec{\beta}.C'\Rightarrow\kappa' \vdash^{ML} x:\forall\vec{\alpha}.C\Rightarrow\kappa$ must look like the above (up to using several subtype steps (which can be achieved by one using transitivity) and repetition of the above sequence (which can be squeezed into one)).

□

The following simple property of the generic instance relation will be very useful:

Lemma 5.7 *If* $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C;A,x:\sigma \vdash^{ML} e:\sigma''}$ *and $\sigma' \sqsubseteq \sigma$ then there exists $\mathcal{T}_2$ such that*

$$\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C;A,x:\sigma' \vdash^{ML} e:\sigma''}$$

[3]That is, S is the identity on all variables not in $\vec{\beta}$

Proof By simple induction on $\mathcal{T}_1$. □

While the above lemma is simple and elegant, we need a more complicated version that also deals with preservation of flow.

Lemma 5.8 *If*

1. $C_1; A \vdash_n^{ML} e' : \kappa_1'$ *is a halbstark instance of* $C_2; A \vdash_n^{ML} e' : \kappa_2'$ *by* S
2. $\sigma_1 = \forall \vec{\alpha}.C_1 \Rightarrow \kappa_1'$ *for some* $\vec{\alpha}$ *where* $\vec{\alpha} \cap FV(A) = \emptyset$ *and* $\sigma_2 = close_A(C_2 \Rightarrow \kappa_2')$
3. $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C; A, x : \sigma_1 \vdash_n^{ML} e : \kappa_1}$

then there exists $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C; A, x : \sigma_2 \vdash^{ML} e : \kappa_2}$
2. $C \vdash^{\forall} \kappa_2 \leq \kappa_1$
3. *Let* $\mathcal{L}$ *be given and let* ϕ *be any environment mapping the domain of* A *to properties. If* $C_1 \vdash S\ell_2 \subseteq \ell_1$ *then for all* $l \in \mathcal{L}$ *we have*

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi_2) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi_1)$$

   where $\phi_2 = \phi[(x, l') \mapsto \ell_2]$ *and* $\phi_1 = \phi[(x, l') \mapsto \ell_1]$ *for all* $l' \in \mathcal{L}$.

4. *Let* $\mathcal{L}$ *be given and let* ϕ *be any environment mapping the domain of* A *to properties. If* $\vec{\alpha}$ *is the empty vector and* $C \vdash S\ell_2 \subseteq \ell_1$ *then for all* $l \in \mathcal{L}$ *we have*

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi_2) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi_1)$$

   where $\phi_2 = \phi[(x, l') \mapsto \ell_2]$ *and* $\phi_1 = \phi[(x, l') \mapsto \ell_1]$ *for all* $l' \in \mathcal{L}$.

Proof By induction over $\mathcal{T}_1$. The only interesting case is (Id).

(Id) Assume

1. $C_1; A \vdash_n^{ML} e' : \kappa_1'$ is a halbstark instance of $C_2; A \vdash_n^{ML} e' : \kappa_2'$ by S
2. $\sigma_1 = \forall \vec{\alpha}.C_1 \Rightarrow \kappa_1'$ for some $\vec{\alpha}$ where $\vec{\alpha} \cap FV(A) = \emptyset$ and $\sigma_2 = close_A(C_2 \Rightarrow \kappa_2')$
3. $\mathcal{T}_1 = \dfrac{C \vdash C_1[\vec{\ell}/\vec{\alpha}]}{C; A, x : \sigma_1 \vdash_n^{ML} x : \kappa_1'[\vec{\ell}/\vec{\alpha}]}$

From 1. we find that

1. $C_1 \vdash SC_2$
2. $A = SA$
3. $C_1 \vdash^{\forall} S\kappa_2' \leq \kappa_1'$

From this we concluded that S is the identity on the free variables of A and hence that it is a substitution on the variables bound by $close_A(C_2 \Rightarrow \kappa_2')$. Furthermore $S' = [\vec{\ell}/\vec{\alpha}] \circ S$ is a substitution on the variables bound by $close_A(C_2 \Rightarrow \kappa_2')$. We then have

1. Since $C \vdash C_1[\vec{\ell}/\vec{\alpha}]$ and $[\vec{\ell}/\vec{\alpha}](C_1) \vdash [\vec{\ell}/\vec{\alpha}](S(C_2))$ we get

$$\mathcal{T}_2 = \frac{C \vdash S'C_2}{C; A, x : \sigma_2 \vdash_n^{ML} x : S'\kappa_2'}$$

2. Since $C \vdash C_1[\vec{\ell}/\vec{\alpha}]$ and $[\vec{\ell}/\vec{\alpha}](C_1) \vdash^{\forall} [\vec{\ell}/\vec{\alpha}](S(\kappa_2')) \leq [\vec{\ell}/\vec{\alpha}](\kappa_1')$ we conclude $C \vdash^{\forall} S'\kappa_2' \leq \kappa_1'[\vec{\ell}/\vec{\alpha}]$
3. Let ϕ be any environment mapping the domain of A to properties. Assume $C_1 \vdash S\ell_2 \subseteq \ell_1$. By definition:

$$\Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi[(x, l) \mapsto \ell_1]) = [\vec{\ell}/\vec{\alpha}](\ell_1)$$

and

$$\Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi[(x, l) \mapsto \ell_2]) = S'(\ell_2)$$

Since $C \vdash C_1[\vec{\ell}/\vec{\alpha}]$ and $[\vec{\ell}/\vec{\alpha}](C_1) \vdash S'(\ell_2) \subseteq [\vec{\ell}/\vec{\alpha}](\ell_1)$ we conclude that for all l we have

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi[(x, l) \mapsto \ell_2]) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi[(x, l) \mapsto \ell_1])$$

4. Assume that $\vec{\alpha}$ is the empty vector. Then $[\vec{\ell}/\vec{\alpha}]$ is the identity. Let ϕ be any environment mapping the domain of A to properties. Assume $C \vdash S\ell_2 \subseteq \ell_1$. By definition:

$$\Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi[(x, l) \mapsto \ell_1]) = \ell_1$$

and

$$\Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi[(x, l) \mapsto \ell_2]) = S'(\ell_2) = S(\ell_2)$$

We find

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi[(x, l) \mapsto \ell_2]) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi[(x, l) \mapsto \ell_1])$$

□

5.2.4 Algorithm and Principality

As only two rules in the syntax directed type system have changed compared to the subtype system, extending the algorithm of figure 3.4 just consists of changing the same two cases:

$$
\begin{array}{ll}
x : & \text{if } A(x) = \forall\vec{\alpha}.C \Rightarrow \kappa \\
 & \text{then } (C[\vec{\alpha'}/\vec{\alpha}], \kappa[\vec{\alpha'}/\vec{\alpha}]) \text{ where } \vec{\alpha'} \text{ is fresh} \\
\text{let } x = e_1 \text{ in } e_2 : & \text{let } (C_1, \kappa) = \mathcal{W}(A, e_1) \\
 & \text{let } \sigma = close_A(C_1 \Rightarrow \kappa) \\
 & \text{let } (C_2, \kappa') = \mathcal{W}((A, x : \sigma), e_2) \\
 & \text{in } (C_2, \kappa')
\end{array}
$$

where the closing function is defined by

$$
close_A(C \Rightarrow \kappa) \;=\; \forall\vec{\alpha}.C \Rightarrow \kappa \text{ where } \alpha \text{ is a sequence of all flow variables free in } C \Rightarrow \kappa \text{ but not in } A.
$$

Using $close_A(C \Rightarrow \kappa)$ we can relate the generic instance relation to the halbstark instance relation as follows:

Lemma 5.9 *If C and C' are constraint sets (that is, contain only constraints of the form $\alpha \subseteq \beta$ and $L \subseteq \alpha$) then:*

1. *If*

   $$C'; A \vdash^{ML} e : \kappa' \text{ is a halbstark instance of } C; A \vdash^{ML} e : \kappa$$

   then

   $$close_A(C \Rightarrow \kappa) \sqsubseteq close_A(C' \Rightarrow \kappa')$$

2. *If*

   $$close_A(C \Rightarrow \kappa) \sqsubseteq close_A(C' \Rightarrow \kappa')$$

   then

   $$C'; A \vdash^{ML} e : \kappa' \text{ is a halbstark instance of } C; A \vdash^{ML} e : \kappa$$

Proof

1. From the definition of halbstark instance, we find a substitution S such that
   (a) $C' \vdash SC$
   (b) $A = S(A)$
   (c) $C' \vdash^{\forall} S(\kappa) \leq \kappa'$

We want to derive that

$$;x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{ML} x : \forall\vec{\beta}.C' \Rightarrow \kappa'$$

where $\vec{\alpha} = FV(C \Rightarrow \kappa) \setminus FV(A)$ and $\vec{\beta} = FV(C' \Rightarrow \kappa') \setminus FV(A)$. This is done by

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{;x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{ML} x : \forall\vec{\alpha}.C \Rightarrow \kappa}\,\mathrm{Id}}{C';x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{ML} x : \forall\vec{\alpha}.C \Rightarrow \kappa}\,\mathrm{weak} \qquad C' \vdash SC}{C';x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{ML} x : S\kappa}\,\forall\text{-E} \qquad C' \vdash^{\forall} S(\kappa) \leq \kappa'}{C';x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{ML} x : \kappa'}\,\mathrm{Sub}}{;x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{ML} x : \forall\vec{\beta}.C' \Rightarrow \kappa'}\,\forall\text{-I}$$

where the last step is allowable since $FV(\forall\vec{\alpha}.C) \subseteq FV(A)$ and no $\beta_i \in \vec{\beta}$ is free in A.

2. By lemma 5.6, S' exists such that $C' \vdash^{\forall} S'(\kappa) \leq \kappa'$ and $C' \vdash S'(C)$ but since S' works only on the variables bound by $close_A(C \Rightarrow \kappa)$ it must be the identity on all variables bound by A and thus $A = S'(A)$ follows.

□

Since *close* clearly is terminating, algorithm $\mathcal{W}$ for ML polymorphic flow analysis must be terminating for the same pairs A, e as algorithm $\mathcal{W}$ for subtype flow analysis. Thus we can use the same definition of proper environments (definition 3.5) and have termination:

Lemma 5.10 *If A is proper then $\mathcal{W}(A, e)$ terminates without failure.*

We now turn to the main theorem concerning principality:

Theorem 5.11 *If $\mathcal{W}(A, e) = (C, \kappa)$ then there exists $\mathcal{T}_1$ such that*

1. $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C; A \vdash_n^{ML} e : \kappa}$ *and*
2. *for any $\mathcal{T}_2$ such that*

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{C'; A \vdash_n^{ML} e : \kappa'}$$

we have that there exists S such that

   (a) *$C'; A \vdash_n^{ML} e : \kappa'$ is a halbstark instance of $C; A \vdash_n^{ML} e : \kappa$ by S and*

   (b) *Let $\mathcal{L}$ be given and let ϕ, ϕ' be two environments mapping free variables of e to properties. Then for all $l \in \mathcal{L}$ we have that $C' \vdash S\phi(x, l) \subseteq \phi'(x, l)$ for all $x \in FV(e)$ implies*

$$C' \vdash S(\Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi')$$

Proof By induction over the structure of e. We give the important cases: (Id) and (let)

(Id) We have

$$\begin{array}{rl} \mathcal{W}(A,x) = & \text{let } \forall\vec{\alpha}.C \Rightarrow \kappa = A(x) \\ & \text{in } (C[\vec{\alpha'}/\vec{\alpha}], \kappa[\vec{\alpha'}/\vec{\alpha}]) \end{array}$$

where $\vec{\alpha'}$ is fresh. Clearly,

$$\frac{C[\vec{\alpha'}/\vec{\alpha}] \vdash C[\vec{\alpha'}/\vec{\alpha}]}{C[\vec{\alpha'}/\vec{\alpha}], A \vdash^{ML} x : \kappa[\vec{\alpha'}/\vec{\alpha}]}$$

Let C', κ' be given such that $C'; A \vdash_n^{ML} x : \kappa'$. I.e.

$$\frac{C' \vdash C[\vec{\ell}/\vec{\alpha}]}{C'; A \vdash_n^{ML} x : \kappa'}$$

where $\kappa' = \kappa[\vec{\ell}/\vec{\alpha}]$. Let $S = [\vec{\ell}/\vec{\alpha'}]$, then

1. $C' \vdash S(C[\vec{\alpha'}/\vec{\alpha}])$
2. $A = SA$
3. $C' \vdash^{\forall} S(\kappa[\vec{\alpha'}/\vec{\alpha}]) \leq \kappa$

since $\alpha_i' \notin FV(A)$ for all α_i' in α' (it is fresh).

Let ϕ and ϕ' be given such that $C' \vdash S\phi(x,l) \subseteq \phi'(x,l)$. But since

$$\begin{array}{rcl} S(\Phi_{\mathcal{L}}(\ \dfrac{C[\vec{\alpha'}/\vec{\alpha}] \vdash C[\vec{\alpha'}/\vec{\alpha}]}{C[\vec{\alpha'}/\vec{\alpha}], A \vdash^{ML} x : \kappa[\vec{\alpha'}/\vec{\alpha}]}, l, \phi)) & = & S([\vec{\alpha'}/\vec{\alpha}](\phi(x,l))) \\ & = & [\vec{\ell}/\vec{\alpha}](\phi(x,l)) \end{array}$$

and

$$\Phi_{\mathcal{L}}(\ \frac{C' \vdash C[\vec{\ell}/\vec{\alpha}]}{C'; A \vdash_n^{ML} x : \kappa'}, l, \phi') = [\vec{\ell}/\vec{\alpha}](\phi'(x,l))$$

we are done

(let) Recall the definition of $\mathcal{W}$ on 'let':

$$\begin{array}{rl} \text{let } x = e_1 \text{ in } e_2 : & \text{let } (C_1, \kappa_1) = \mathcal{W}(A, e_1) \\ & \text{let } \sigma = close_A(C_1 \Rightarrow \kappa_1) \\ & \text{let } (C_2, \kappa_2) = \mathcal{W}((A, x : \sigma), e_2) \\ & \text{in } (C_2, \kappa') \end{array}$$

We assume that $\mathcal{T}_3$ is given such that

$$\mathcal{T}_3 = \frac{\mathcal{T}_4 \quad \mathcal{T}_5}{C_3; A \vdash_n^{ML} \text{let } x = e_1 \text{ in } e_2 : \kappa_3}$$

Without loss of generality we can assume

$$\mathcal{T}_4 = \frac{\mathcal{T}_4'}{C_4; A \vdash_n^{ML} e_1 : \kappa_4} \quad \text{and} \quad \mathcal{T}_5 = \frac{\mathcal{T}_5'}{C_3; A, x : \forall \vec{\alpha}.C_4 \Rightarrow \kappa_4 \vdash_n^{ML} e_2 : \kappa_3}$$

where $\vec{\alpha} \cap (FV(A) \cup FV(C_3)) = \emptyset$.

Let ϕ, ϕ' be two environments mapping free variables of let $x = e_1$ in e_2 to properties.

By induction we find that there exists $\mathcal{T}_1$ such that:

1. $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C_1; A \vdash_n^{ML} e : \kappa_1}$ and
2. there exists S_1 such that
   (a) $C_4; A \vdash_n^{ML} e_1 : \kappa_4$ is a halbstark instance of $C_1; A \vdash_n^{ML} e : \kappa_1$ by S_1 and
   (b) Let $\phi_1 = \phi \mid_{FV(e_1)}$ and $\phi_1' = \phi' \mid_{FV(e_1)}$. Then for all $l \in$ Destructors(e_1) we have that $C_4 \vdash S_1\phi_1(y, l) \subseteq \phi_1'(y, l)$ for all $y \in FV(e_1)$ implies

$$C_4 \vdash S_1(\Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi_1)) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_4, l, \phi_1')$$

We can apply lemma 5.8 to find

$$\mathcal{T}_6 = \frac{\mathcal{T}_6'}{C_3; A, x : close_A(C_1 \Rightarrow \kappa_1) \vdash_n^{ML} e_2 : \kappa_4}$$

where

1. $C_3 \vdash^{\forall} \kappa_4 \leq \kappa_3$
2. Let ϕ be any environment mapping the domain of A to properties. If $C_4 \vdash S_1\ell_2 \subseteq \ell_1$ then for all $l \in$ Destructors(e_2) we have

$$C_3 \vdash \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi[(x, l) \mapsto \ell_2]) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi[(x, l) \mapsto \ell_1])$$

By the induction hypothesis we find that there exists $\mathcal{T}_2$ such that

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C_2; A, x : close_A(C_1 \Rightarrow \kappa_1) \vdash_n^{ML} e_2 : \kappa_2}$ and
2. there exists S_2 such that
   (a) $C_3; A, x : close_A(C_1 \Rightarrow \kappa_1) \vdash_n^{ML} e_2 : \kappa_4$ is a halbstark instance of $C_2; A, x : close_A(C_1 \Rightarrow \kappa_1) \vdash_n^{ML} e_2 : \kappa_2$ by S_2 and
   (b) Let $\phi_2 = \phi[(x, l) \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)]$ $\phi_2' = \phi'[(x, l) \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi')]$. Assume $C_3 \vdash S_2\phi_2(y, l) \subseteq \phi_2'(y, l)$ for all $y \in FV(e_2)$. Then for all $l \in$ Destructors(e_2) we have

$$C_3 \vdash S_2(\Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi_2)) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_2')$$

We can now construct $\mathcal{T}$ such that

1. $\mathcal{T} = \dfrac{\mathcal{T}_1 \quad \mathcal{T}_2}{C_2; A \vdash_n^{ML} \text{let } x = e_1 \text{ in } e_2 : \kappa_2}$ and
2. (a) $C_3; A \vdash_n^{ML} \text{let } x = e_1 \text{ in } e_2 : \kappa_4$ is a halbstark instance of $C_2; A \vdash_n^{ML} \text{let } x = e_1 \text{ in } e_2 : \kappa_2$ by S_2 and
   (b) Assume $C_3 \vdash S_2\phi(y,l) \subseteq \phi'(y,l)$ for all $y \in FV(\text{let } x = e_1 \text{ in } e_2)$. Then for all $l \in \text{Destructors}(\text{let } x = e_1 \text{ in } e_2)$ we have

$$C_3 \vdash S_2(\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi)) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi')$$

1. is OK. 2(a) is OK. For 2(b) we assume $C_3 \vdash S_2\phi(y,l) \subseteq \phi'(y,l)$ for all $y \in FV(\text{let } x = e_1 \text{ in } e_2)$. This implies that $C_3 \vdash S_2\phi_2(y,l) \subseteq \phi'(y,l)_2$ for all variables $y \in FV(e_2)$. Then for all $l \in \text{Destructors}(e_2)$ we have

$$C_3 \vdash S_2(\Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi_2)) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_2')$$

By definition $\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi_2)$. We miss

$$C_3 \vdash \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_2') \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi')$$

We have by definition that

$$\Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi') = \Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi'[(x,l) \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_4, l, \phi')])$$

which leaves us with

$$\begin{aligned} C_3 \vdash \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi'[(x,l) \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi')]) \\ \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi'[(x,l) \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_4, l, \phi')]) \end{aligned}$$

which we showed above

□

From the above lemma and theorem as well as theorem 5.1, we conclude that every term has a principal type (under halbstark instance).

Note that we could have proven the existence of principal types for the sub-type based flow analysis of chapter 3 under the halbstark instance relation. This would have been a stronger result (as it would imply principal types under lazy instance) and would have been slightly simpler to prove.

5.2.5 Accelerated Algorithm and Minimality

Consider the following expression:

```
let x_0 = e
 in  let x_1 = if y then x_0 else x_0
      in  let ···
           in  let x_n = if y then x_{n-1} else x_{n-1}
                in if y then x_n else x_n
```

Assume that analysing e yields constraints set C_0. Since x_0 will be polymorphic, we will make two instances of C_0. This will be performed a number of times proportional to the size of the program, and thus the resulting constraint set will have a size exponential in the size of the program.

This shows that a naive implementation of the above algorithm will lead to exponential worst-case behaviour. It is not impossible that smart representations of constraint set might improve the situation, but instead of trying to mend a basically flawed (from an algorithmic point of view) algorithm, we will attack the heart of the problem: the exploding size of constraint sets. The idea is to reduce the size of constraint sets inferred so they will be proportional to the size of the standard type instead of the size of the expression.

We will proceed as we have done in previous chapters to find minimal typings. In contrast to these chapters, however, we are not only interested in minimal types but also in an efficient way of computing them. The method of reducing constraint sets will thus not only be applied after a result has been found but at every let-bound expression. This will reduce the constraint sets making the duplication by different instantiations less costly.

Due to the identity rule (Id) we can assume that constraint sets contain no constraints of the form $\alpha \subseteq \alpha$. To be more precise, if C' is the result of removing all constraints of the form $\alpha \subseteq \alpha$ from a constraint set C, then $C; A \vdash^{ML} e : \kappa$ and $C'; A \vdash^{ML} e : \kappa$ are equivalent.

Definition 5.12 *Assume* $A = \{x_1 : \forall\vec{\alpha_1}.C_1 \Rightarrow \kappa_1, \cdots, x_n : \forall\vec{\alpha_n}.C_n \Rightarrow \kappa_n\}$. *A flow variable occurring free in* A *or* κ *occurs* positively *(*negatively*) in* $C; A \vdash e : \kappa$ *if it occurs positively (negatively) in* $\kappa_1 \to \cdots \to \kappa_n \to \kappa$. *Variables not occurring free in* A *or* κ *are said to be* neutral.

All variables occurring in C will occur positively, negatively, positively and negatively or be neutral. In chapter 3 we proved that we could remove all non-negative variables from a judgement $C; A \vdash^{\leq} e : \kappa$ and arrive at an equivalent judgement (lemma 3.10). Recall, however, that equivalence was using the *lazy* instance relation — we can indeed prove a similar lemma for polymorphic flow analysis using lazy instance, but it is not true for halbstark instance.

Fortunately, we will only need the lemma for neutral variables (as in chapters 2 and 3:

Lemma 5.13 *Let* α *be a flow variable with only neutral occurrences in* $C; A \vdash^{ML} e : \kappa$.

Let $\ell_1 \subseteq \alpha, \cdots, \ell_n \subseteq \alpha$ *be all inequalities in* C *with* α *on the right-hand side. We delete these from* C *and replace every inequality* $\alpha \subseteq \beta$ *in* C *by* $\ell_1 \subseteq \beta, \cdots, \ell_n \subseteq \beta$ *and call the resulting constraint set* C'.

Then $C; A \vdash^{ML} e : \kappa$ *and* $C'; A \vdash^{ML} e : \kappa$ *are equivalent (halbstark instances of each other).*

Proof Remark that $C \vdash \bigsqcup_i \ell_i \subseteq \alpha$.

First, $C; A \vdash^{ML} e : \kappa$ is an instance of $C'; A \vdash^{ML} e : \kappa$ since

1. $C \vdash C'$,
2. $A = A$, and
3. $C \vdash^{\forall} \kappa \leq \kappa$

where 1. is proven by transitivity and 2. and 3. are trivial.

To see that $C'; A \vdash^{ML} e : \kappa$ is an instance of $C; A \vdash^{ML} e : \kappa$, let $S = \{\alpha \mapsto \bigsqcup_i \ell_i\}$. Now

1. $C' \vdash S(C)$,
2. $A = S(A)$, and
3. $C' \vdash^{\forall} S(\kappa) \leq \kappa$

where 1. follows from $C' \vdash \ell_j \leq \bigsqcup_i \ell_i$ for all j and $C' \vdash \bigsqcup_i \ell_i \leq \beta$. Points 2. and 3. are trivial since α does not occur in A or κ.

□

Lemma 3.11 generalises to the polymorphic system:

Lemma 5.14 *Let α be any flow variable then $C, L \subseteq \alpha, L' \subseteq \alpha; A \vdash^{ML} e : \kappa$ and $C, L \cup L' \subseteq \alpha; A \vdash^{ML} e : \kappa$ are equivalent.*

By applying Lemma 5.13 to all neutral variables in a judgement and applying lemma 5.14 exhaustively, we arrive at the following theorem:

Theorem 5.15 *For every flow judgement $C; A \vdash^{ML} e : \kappa$ there exists an equivalent judgement $C'; A \vdash^{ML} e : \kappa$ where C' contains only flow variables α occurring free in A or κ and for each α there is only one constraint of the form $L \subseteq \alpha$ in C'.*

By applying lemma 5.13 to neutral variables only, the resulting judgement does not include any $\sqcup$'s. In particular, no types in the environment nor in the resulting type will contain $\sqcup$ and hence the algorithm for *constraints* need not be changed.

Note, that applying theorem 5.15 does not affect the flow computed by the derivations.

If we change the algorithm slightly to account for the above, we get:

$$
\begin{array}{ll}
x : & \text{if } A(x) = (\forall \vec{\alpha}.C \Rightarrow \kappa) \\
 & \text{then } (C[\vec{\alpha'}/\vec{\alpha}], \kappa[\vec{\alpha'}/\vec{\alpha}]) \text{ where } \vec{\alpha'} \text{ is fresh} \\
\text{let } x = e_1 \text{ in } e_2 : & \text{let } (C_1, \kappa) = \mathcal{W}(A, e_1) \\
 & \text{let } C_1' = \mathit{elim}(\mathrm{FV}(A) \cup \mathrm{FV}(\kappa), C_1) \\
 & \text{let } \sigma = \mathit{close}_A(C_1' \Rightarrow \kappa) \\
 & \text{let } (C_2, \kappa') = \mathcal{W}((A, x : \sigma), e_2) \\
 & \text{in } (C_2, \kappa')
\end{array}
$$

where $elim(V, C)$ is defined to be a constraint set C' obtained by eliminating all variables not in V (according to lemma 5.13) and merging all $L_i \subseteq \alpha$ constraints with identical right-hand sides according to lemma 5.14.

5.2.6 Complexity

Let n be the size of the analysed program with explicit standard types on all subexpressions (eg. $(\lambda x.(\text{if True} : \text{Bool then } x : \text{Bool else } y : \text{Bool}) : \text{Bool}) : \text{Bool} \to \text{Bool}$). First we establish the maximum size of a constraint set C resulting from a call to $\mathcal{W}$.

Lemma 5.16 *If $(C, \kappa') = \mathcal{W}(A, e)$ then the number of constraints in C is bounded by $\mathcal{O}(n^3)$ and the number of variables and constants in C is bounded by $\mathcal{O}(n^2)$.*

Proof Consider the size of $C_1' = elim(\text{FV}(A) \cup \text{FV}(\kappa), C)$. By theorem 5.15 C_1' contains only flow variables occurring free in A or κ. The number of these variables must be bounded by n and therefore the number of constraints of the form $\alpha \subseteq \beta$ is bounded by n^2 and the number of constraints $L \subseteq \alpha$ is bounded by n. Thus the number of constraints in C_1' must be bounded by n^2 and the number of "nodes" (variables and constants) is bounded by n.

As there can be at most n occurrences of let-bound variables in e, we conclude that the upper bound on the number of constraints in C is cubic (since all other constructs generate a constant number of constraint plus the constraints of subexpressions). Similarly the number of nodes must be bounded by n^2. □

Next we show a polynomial upper bound on the maximal running time of the $elim()$ function.

Lemma 5.17 *All calls to $elim(F, C)$ are bounded by $\mathcal{O}(n^6)$ for every set F of flow variables.*

Proof Lemma 5.13 passes over all constraints in C and removes and adds constraints. If the constraint set is viewed as a graph, the number of nodes decreases by one in each application since a variable is deleted. The maximal number of constraints added by one application of the lemma is thus $\mathcal{O}(n^4)$ since the number of nodes is $\mathcal{O}(n^2)$. Lemma 5.13 can be applied at most $\mathcal{O}(n^2)$ times, thus we find the complexity of this part of $elim(F, C)$ to be $\mathcal{O}(n^6)$.

The second part (lemma 5.14) is bounded by the number of constraints and is thus $\mathcal{O}(n^4)$.

□

It would be tempting to use the upper bound $\mathcal{O}(n^3)$ on the number of constraints in C but this might not be maintained by successive applications of lemma 5.13.

Finally the main complexity result of the accelerated algorithm.

Theorem 5.18 *Accelerated algorithm $\mathcal{W}$ computes a principal type of e in time $\mathcal{O}(n^7)$.*

Proof There can be at most n let-expressions in e, thus at most n calls to *elim*(). Making the final result minimal does not add anything (just one extra application of *elim*()). □

This complexity seems prohibitive for implementations — we are, however, not sure that the bound is tight. Furthermore, we believe that the algorithm will be well-behaved in practice. E.g. consider the proof of lemma 5.16: we found a bound of n on the number of variables in the constraint set C resulting from calls to *elim* and concluded that the number of constraints in C was bounded by n^2. This, however, is only the case when the constraint set is dense; we conjecture that it will often be sparse in practice. Similar considerations might well lead to better practical behaviour. Finally, it is very likely that more efficient algorithms exist [DHM95b].

5.2.7 Soundness

We can prove strong subject reduction for the ML-polymorphic system. We first prove the usual substitution lemma:

Lemma 5.19 (Substitution lemma) *If*

$$\mathcal{T}_1 = \frac{\mathcal{T}_1'}{C; A, x : \kappa \vdash_n^{ML} e : \kappa'}$$

and

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{C; A \vdash_n^{ML} e' : \kappa}$$

then there exists $\mathcal{T}_3$ such that

1. $\mathcal{T}_3 = \dfrac{\mathcal{T}_3'}{C; A \vdash_n^{ML} e[e'/x] : \kappa'}$ *and*
2. *For any ϕ and $\mathcal{L}$ and all $l \in \mathcal{L}$ we have*

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi') \sqcup \Phi(\mathcal{T}_2, l, \phi)$$

where $\phi' = \phi[(x, l') \mapsto \{\}]$ for all $l' \in \mathcal{L}$

Proof The lemma follows by induction on the structure of $\mathcal{T}_1$. Since we assume that x is monomorphic, the proof is identical to the proof of lemma 3.13 □

Lemma 5.19 does the hard work for most cases of our strong subject reduction theorem. In this polymorphic case, however, it does not suffice for the let-case, so we prove the following variant of the substitution lemma:

Lemma 5.20 *If*

$$\mathcal{T}_1 = \frac{\mathcal{T}_1'}{C'; A, x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{ML} e : \kappa'}$$

and

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{C; A \vdash_n^{ML} e' : \kappa}$$

then there exists $\mathcal{T}_3$ *such that*

1. $\mathcal{T}_3 = \dfrac{\mathcal{T}_3'}{C; A \vdash_n^{ML} e[e'/x] : \kappa'}$ *and*
2. *For any* ϕ *and* $\mathcal{L}$ *and all* $l \in \mathcal{L}$ *we have*

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi')$$

where $\phi' = \phi[(x, l') \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_2, l', \phi)]$ *for all* l'

Proof The proof is by induction over $\mathcal{T}_1$. The only interesting case is (Id): Assume

$$\mathcal{T}_1 = \frac{C' \vdash C[\vec{\ell}/\vec{\alpha}]}{C'; A, x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{ML} x : \kappa[\vec{\ell}/\vec{\alpha}]}$$

and $\kappa' = \kappa[\vec{\ell}/\vec{\alpha}]$.

Since $C' \vdash C[\vec{\ell}/\vec{\alpha}]$ it is a trivial induction on $\mathcal{T}_2$ to show the existence of

$$\mathcal{T}_3 = \frac{\mathcal{T}_3'}{C'; A \vdash_n^{ML} e' : \kappa[\vec{\ell}/\vec{\alpha}]}$$

and that $\Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi) = [\vec{\ell}/\vec{\alpha}](\Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi))$ but since the latter equals $\Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi')$ by definition, we are done.

□

Theorem 5.21 (Subject Reduction) *If* $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C; A \vdash_n^{ML} e_1 : \kappa}$ *and* $e_1 \longrightarrow e_2$ *then there exists* $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C; A \vdash_n^{ML} e_2 : \kappa}$ *and*

2. *For any* ϕ *and* $\mathcal{L}$ *and all* $l \in \mathcal{L}$ *we have*

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)$$

and if $l \in \text{Destructors}(e_1)$ *consumes* $l' \in \text{Contructors}(e_1)$ *then for any* ϕ *and* $\mathcal{L}$ *(where* $l, l' \in \mathcal{L}$*) then*

$$C \vdash \{l'\} \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)$$

Proof The main cases follows from lemmas 5.19 and 5.20. Note, that the formulation with Φ instead of $\mathcal{F}$ is necessary for the inductive (context) case. □

Soundness of flow functions computed from inference trees follows as usual (the result remains true by definition, if the derivation $\mathcal{T}$ is not syntax directed):

Corollary 5.22 *Let* $\mathcal{T}$ *be any derivation for* e *and let* $C; A \vdash_n^{ML} e : \kappa$ *be its conclusion. Then* $C; \mathcal{F}_{\mathcal{T}} \models e$.

5.2.8 Invariance under Transformation

We will prove the subject expansion property for 'let'. This implies that unfolding let-expressions cannot improve the result of this flow analysis — explicit or implicit unfolding of definitions (known as *polyvariance*) has been used in program analysis for improving results. This result shows that we can obtain equally good results without unfolding definitions. When we add fix-polymorphism in the next section, a similar result will be proved for unfolding recursive definitions.

Just as the statement of soundness implied preservation of flow, this (partial) *subject expansion* property states that flow is preserved.

Recall that $x^{(i)}$ denotes the i'th occurrence of x in e.

Lemma 5.23 *Let* e *be an expression with* n *occurrences of a variable* x*, and let* $e_1, \cdots, e_n$ *be* n *expressions where no free variable of* e_i *is bound by* e*. If*

$$\mathcal{T}_{e[e_i/x^{(i)}]} = \frac{\mathcal{T}'_{e[e_i/x^{(i)}]}}{C; A \vdash_n^{ML} e[e_i/x^{(i)}] : \kappa}$$

then there exists $\kappa_1 \cdots \kappa_n$ *and* $\mathcal{T}_e, \mathcal{T}_1, \cdots, \mathcal{T}_n$ *such that*

1. $\mathcal{T}_i = \dfrac{\mathcal{T}'_i}{C; A \vdash_n^{ML} e_i : \kappa_i}$

2. $\mathcal{T}_e = \dfrac{\mathcal{T}'_e}{C; A, x_1 : \kappa_1, \cdots, x_n : \kappa_n \vdash_n^{ML} e[x_i/x^{(i)}] : \kappa}$ *and*

3. *For any ϕ and $\mathcal{L}$ and all $l \in \mathcal{L}$ we have*

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_{e[e_i/x^{(i)}]}, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_e, l, \phi) \sqcup \bigsqcup_i \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi)$$

Proof By induction on the inference tree $\mathcal{T}_{e[e_i/x^{(i)}]}$.

□

Theorem 5.24 (Invariance under let-conversion) *If*

$$\mathcal{T}_{e'[e/x]} = \frac{\mathcal{T}'_{e[e/x]}}{C; A \vdash_n^{ML} e'[e/x] : \kappa}$$

then there exists $\mathcal{T}_{let}$ such that

1. $\mathcal{T}_{let} = \dfrac{\mathcal{T}'_{let}}{C; A \vdash_n^{ML} \text{let } x = e \text{ in } e' : \kappa_{let}}$
2. $C \vdash^{\forall} \kappa_{let} \leq \kappa$
3. *For any ϕ and $\mathcal{L}$ and all $l \in \mathcal{L}$ we have*

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_{let}, l, \phi) \leq \Phi_{\mathcal{L}}(\mathcal{T}_{e'[e/x]}, l, \phi)$$

Proof Let ϕ and $\mathcal{L}$ be given. Assume $\phi(x) = \emptyset$ (since x not free in $e'[e/x]$ or let $x = e$ in e', this assumption is valid). Assume e' contains n occurrences of x. By lemma 5.23 there exist $\kappa_1, \cdots, \kappa_n$ and $\mathcal{T}_{e'}, \mathcal{T}_1, \cdots, \mathcal{T}_n$ such that

1. $\mathcal{T}_i = \dfrac{\mathcal{T}'_i}{C; A \vdash_n^{ML} e : \kappa_i}$
2. $\mathcal{T}_{e'} = \dfrac{\mathcal{T}'_{e'}}{C; A, x_1 : \kappa_1, \cdots, x_n : \kappa_n \vdash_n^{ML} e'[x_i/x^{(i)}] : \kappa}$ and
3. For all $l \in \mathcal{L}$ we have

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_{e'[e/x]}, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_{e'}, l, \phi) \sqcup \bigsqcup_i \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi)$$

By theorem 5.11 we know that we can derive a principal typing

$$\mathcal{T} = \frac{\mathcal{T}'}{C'; A \vdash_n^{ML} e : \kappa'}$$

for e under A such that for every i there exists S_i where

1. $C; A \vdash_n^{ML} e : \kappa_i$ is a halbstark instance of $C'; A \vdash_n^{ML} e : \kappa'$ under S_i and

2. Let $\phi'_i = S_i\phi$. Since by definition $C \vdash S_i\phi(y,l) \leq \phi'(y,l)$ for all y and $l \in \mathcal{L}$ we have

$$C \vdash S_i(\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi)) \leq \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi'_i)$$

Every derivation α-equivalent to $\mathcal{T}$ is also principal with the same properties — thus, without loss of generality, we can assume that $\mathrm{Ran}(\phi) \cap (FV(C') \cup FV(\kappa')) = \emptyset$. This implies that $\phi'_i = \phi$.

By repeated application of lemma 5.8 (special case where C_1 and α of this lemma are both empty) we find the existence of $\mathcal{T}_2$ such that

1. $\mathcal{T}_2 =$

$$\frac{\mathcal{T}'_2}{C; A, x_1 : close_A(C' \Rightarrow \kappa'), \cdots, x_n : close_A(C' \Rightarrow \kappa') \vdash^{ML}_n e'[x_i/x^{(i)}] : \kappa_{let}}$$

2. $C \vdash^{\forall} \kappa_{let} \leq \kappa$

3. We have
$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi_2) \leq \Phi_{\mathcal{L}}(\mathcal{T}_{e'}, l, \phi_1)$$
where $\phi_1 = \phi[(x_i, l') \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi'_i)]$ and $\phi_2 = \phi[(x_i, l') \mapsto \Phi_{\mathcal{L}}(\mathcal{T}, l, \phi)]$ for all $l' \in \mathcal{L}$. We used the fact that $C \vdash S_i\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi) \leq \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi'_i)$.

Let $\mathcal{T}_{let} =$

$$\frac{\dfrac{\mathcal{T}'}{C; A \vdash^{ML}_n e : \kappa'} \qquad \dfrac{\mathcal{T}''_2}{C; A, x : close_A(C' \Rightarrow \kappa') \vdash^{ML}_n e' : \kappa_{let}}}{C; A \vdash^{ML}_n \text{let } x = e \text{ in } e' : \kappa_{let}}$$

where $\mathcal{T}''_2$ is identical to $\mathcal{T}'_2$ except the assumptions on $x_1, \cdots, x_n$ has been replaced by x. Now

$$\Phi_{\mathcal{L}}(\mathcal{T}_{let}, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi_2)$$

and

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi_2) \leq \Phi_{\mathcal{L}}(\mathcal{T}_{e'}, l, \phi_1)$$

It is not hard to see that

$$\begin{aligned} C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_{e'}, l, \phi_1) &= \Phi_{\mathcal{L}}(\mathcal{T}_{e'}, l, \phi) \sqcup \bigsqcup_i \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi'_i) \\ &= \Phi_{\mathcal{L}}(\mathcal{T}_{e'}, l, \phi) \sqcup \bigsqcup_i \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi) \end{aligned}$$

We have proved

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_{let}, l, \phi) \leq \Phi_{\mathcal{L}}(\mathcal{T}_{e'[e/x]}, l, \phi)$$

□

Corollary 5.25 *If*

$$\mathcal{T}_1 = \frac{\mathcal{T}_1'}{C; A \vdash_n^{ML} C[e'[e/x]] : \kappa_1}$$

then there exists $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C; A \vdash_n^{ML} C[\text{let } x = e \text{ in } e'] : \kappa_2}$
2. $C \vdash^{\forall} \kappa_2 \leq \kappa_1$
3. *For all* $l \in \text{Destructors}(C[e'[e/x]])$ *we have*

$$C \vdash \mathcal{F}_{\mathcal{T}_1}(l) = \mathcal{F}_{\mathcal{T}_2}(l)$$

Proof We prove the property where we replace *3.* by

For any ϕ and $\mathcal{L}$ and all $l \in \mathcal{L}$ we have

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi)$$

The proof is by induction over the structure of C. The base case is theorem 5.24 and the induction cases are straightforward.

□

5.3 Polymorphic recursion

The idea of polymorphic recursion is to allow type schemes not only in let-bound expressions but also in fix-bound expressions.

Example 5.26 Consider the following (admittedly strange) definition of the identity function:

$$\text{fix } f.((\lambda^{l_1} y.\lambda^{l_2} x.x)@^{l_3}(f@^{l_4}\text{True}^{l_5}, f@^{l_6}\text{True}^{l_7})^{l_8})$$

Here ML-polymorphism doesn't help us to give this term a better type than:

$$\forall\alpha.\{l_5, l_7\} \subseteq \alpha \Rightarrow \text{Bool}^{\alpha} \rightarrow^{\{l_2\}} \text{Bool}^{\alpha}$$

The point is that *all* recursive occurrences have to have the same type and this also has to be the same as the resulting type. If we lift this restriction and allow type schemes in recursive definitions, we can get the following more general type:

$$\forall\alpha.\text{Bool}^{\alpha} \rightarrow^{\{l_2\}} \text{Bool}^{\alpha}$$

□

For standard type inference, this extension has been studied by Mycroft [Myc84] but was later shown to be undecidable. In program analysis, adding polymorphic recursion over annotations has proven to be both decidable and desirable:

- In region inference (Tofte and Talpin [TT94])
- Dimension inference (Rittri [Rit95])
- In binding-time analysis (Henglein and the present author [HM94] and Dussart, Henglein and the present author [DHM95a])

(in the first case it is yet unknown whether principal types are computable). We will show that the extension leads to a decidable and even polynomial time algorithm for flow analysis.

We only have to change one rule of the ML-polymorphic system to accommodate polymorphic recursion:

$$\text{fix}\ \frac{C; A, x : \sigma \vdash^{fix} e : \sigma}{C; A \vdash^{fix} \text{fix}\, x.e : \sigma}$$

We will now follow the same road that we have followed for subtypes and ML-polymorphic types: (1) Make the system syntax directed, (2) Define an algorithm computing principal types, (3) Compute the minimal type (as with ML-polymorphism, use this to speed up the algorithm). There is an important difference: it will not be clear that a straightforward algorithm will always terminate. Thus (3) is necessary not only to speed up the algorithm of (2) but also to make sure it terminates (when the algorithm based on (3) has been shown to be terminating it will follow that so was the algorithm based on (2)). Furthermore, the algorithm using constraint reduction has in the fix-polymorphic case room for further improvement.

In the next subsection we will make the system syntax directed and we will then step back and consider polymorphic recursion in general and in particular sketch Mycroft's original argument for principal types. His argument suggests a (semi-)algorithm which will be the basis for our algorithm.

5.3.1 Syntax Directed System

We only need one change compared to the syntax directed ML-polymorphic system (figures 3.3 and 5.3):

$$\text{fix}\ \frac{C; A, x : \forall \vec{\alpha}.C \Rightarrow \kappa \vdash_n^{fix} e : \kappa' \quad C \vdash^{\forall} \kappa' \leq \kappa \quad C' \vdash C[\vec{\ell}/\vec{\alpha}]}{C'; A \vdash_n^{fix} \text{fix}\, x.e : \kappa[\vec{\ell}/\vec{\alpha}]}\ (*)$$

where $(*)$ means $(\vec{\alpha} \cap (\text{FV}(A) \cup \text{FV}(C'))) = \emptyset$.

Theorem 5.27 *Type system $\vdash_n^{fix}$ is sound and complete w.r.t. $\vdash^{fix}$:*

Soundness *If $C; A \vdash_n^{fix} e : \kappa$ then $C; A \vdash^{fix} e : \kappa$*

Completeness *If $C; A \vdash^{fix} e : \kappa$ then there exists κ' such that $C; A \vdash_n^{fix} e : \kappa'$ and $C \vdash \kappa' \leq \kappa$.*

Proof The proof of soundness proceeds by induction over the derivation of $C; A \vdash_n^{fix} e : \kappa$

The proof of completeness proceeds by induction over the derivation of $C; A \vdash^{fix} e : \kappa$

□

We extend the definition of flow functions by adding the following case to the definition of Φ:

$$\Phi_{\mathcal{L}}(\mathrm{fix}\frac{\mathcal{T}}{C'; A \vdash_n^{fix} \mathrm{fix}\, x.e : \kappa[\vec{\ell}/\vec{\alpha}]}, l, \phi) = [\vec{\ell}/\vec{\alpha}](\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi'))$$

where ϕ' is the least solution to the equation

$$\phi' = \phi[(x, l') \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi')]$$

for all $l' \in \mathcal{L}$.

The domain and range of ϕ' is finite and if ϕ_1 and ϕ_2 and solutions to $\phi' = \phi[(x, l') \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi')]$ then so is $\phi_1 \sqcup \phi_2$, hence the definition is well-defined.

As in the ML-polymorphic flow analysis of the previous section, a derivation contains more information than is expressed by the sticky interpretation. This information can in certain settings (such as binding-time analysis [HM94]) be useful.

5.3.2 Polymorphic Recursion Revisited: Kleene-Mycroft Iteration

This subsection will briefly recapitulate Mycroft's original method for computing principal types for fix-expressions under polymorphic recursion. The description stems from the work on polymorphic binding-time analysis by Dussart, Henglein and the present author [DHM95a].

Consider an ML-polymorphic type system [Mil78, DM82, Dam84], extended with a rule for *polymorphic recursion* [Myc84]:

$$\frac{A, f : \sigma \vdash e : \sigma}{A \vdash \mathrm{fix}\, f.e : \sigma}$$

where σ can be a *type scheme (polymorphic type)*; that is, a type of the form $\forall\vec{\alpha}.\tau$, where τ is a simple type.[4]

[4]This typing calculus has also been termed *Milner-Mycroft Calculus* [Hen88].

Mycroft has shown that this strengthened type system has the principal typing property: every expression e typable under assumptions A has a principal type $\sigma = \forall\vec{\alpha}.\tau$ such that if $A \vdash e : \forall\vec{\beta}.\tau'$ then $\sigma' = \forall\vec{\beta}.\tau'$ is a *generic instance* of σ; that is, $\vec{\beta} \cap FV(\sigma) = \emptyset$ and $\tau' = S(\tau)$ for some substitution S with domain contained in $\vec{\alpha}$. His proof also yields a "natural" algorithm for computing the principal type.

The algorithm and its correctness are corollaries of an elegant general argument couched in terms of notions of domain theory. Let us recapitulate Mycroft's argument since it is very general and since it is at the heart of our adaptation to polymorphic flow analysis, which in addition to Mycroft's polymorphic recursion has qualified types.

Mycroft shows that the set of α-equivalence classes of type schemes $\forall\vec{\alpha}.\tau$, together with an artificial top element $\top$ forms a cpo (complete partial order) under the generic instance ordering $\sqsubseteq$, where, with the exception of $\top$, all elements of the cpo are *compact*[5].

Let $\mathrm{fix}\, f.e$ be a fix-expression. For now, let us assume that it is closed and contains no other fix-expressions. Mycroft defines a function $F_{A,e}$ on α-equivalence classes of type schemes, by

$$F_{A,e}(\sigma) = \sigma' \Leftrightarrow A, f : \sigma \vdash e : \sigma'$$

where σ' is the *principal type* of e under $A, f : \sigma$. If $\sigma = \top$ or if e has no type under $A, f : \sigma$, then $F_{A,e}(\sigma) = \top$. He shows that $F_{A,e}$ is well-defined and continuous[6].

From these general properties, all desired results follow: the principal typing property as well as a natural algorithm for computing principal types of fix-expressions. By standard fixed-point theory for cpo's we know that $F_{A,e}$ has a least fixed point σ_p. If $\sigma_p = \top$ then $\mathrm{fix}\, f.e$ is not typable. If $\sigma_p \neq \top$, then:

1. σ_p is a type of $\mathrm{fix}\, f.e$ since $A, f : \sigma_p \vdash e : \sigma_p$; this follows from the fact that σ_p is a fixed point;

2. σ_p is a *principal* type of $\mathrm{fix}\, f.e$; this follows from the fact that any other type of $\mathrm{fix}\, f.e$ is a fixed-point of $F_{A,e}$, and σ_p is the least fixed point with respect to the generic instance ordering $\sqsubseteq$;

3. σ_p can be computed by constructing a *Kleene sequence*; that is, there exists $k < \omega$ such that $\sigma_p = F^k_{A,e}(\bot)$; this follows from the fact that σ_p is compact.

In summary, if $\mathrm{fix}\, f.e$ has a type under assumptions A, then it has a *principal* type that can be computed by iterating $F_{A,e}$ on $\bot = \forall\alpha.\alpha$ until

[5]Let D be a cpo. An element $x \in D$ is *compact* if, for every directed collection M such that $x \sqsubseteq \bigsqcup M$, there is some $y \in M$ such that $x \sqsubseteq y$ [Gun92].

[6]It is actually enough to know that $F_{A,e}$ is monotonic.

we obtain a fixed point; that is until $F^k_{A,e}(\bot) = F^{k+1}_{A,e}(\bot)$. Recall that equality here means α-equivalence. We call the computation of $F^k_{A,e}(\bot)$ *Kleene-Mycroft Iteration (KMI).*

For expressions containing nested fix-expressions — that is, fix-expressions inside of fix-expressions — Algorithm $\mathcal{W}$ [Mil78] is used to compute principal types in conjunction with Kleene-Mycroft Iteration for fix-expressions.

5.3.3 Algorithm I: Principality

Dussart, Henglein and the present author have shown that polymorphic recursion for binding-time analysis was not only decidable but also polynomial time computable [DHM95a]. We will follow the general approach of that paper. There is a fundamental difference between binding-time analysis and flow analysis: in binding-time analysis we have a two-point domain $\{\mathsf{S}, \mathsf{D}\}$, whereas in flow analysis the domain is only bounded by the size of the program and even exponential in size of the program (since we have sets of labels). This makes it more difficult to bound our version of Kleene-Mycroft sequences.

This subsection presents the first attempt at an algorithm. The algorithm will allow us to state principality and minimality results, but its complexity will be forbidding (it will not even be obvious that it is terminating). The following subsections will mend this problem.

The domain theory employed can be found in any text book on the subject, eg. [Gun92].

Definition 5.28 *For any type t, let $\kappa \in \mathcal{K}^\forall(t)$ be a flow type where all annotations are distinct free type variables $\vec{\alpha}$. Define $\bot_t = \forall\vec{\alpha}.\kappa$*

Lemma 5.29 *Let $\mathcal{L}$ be a finite set*[7]*. For any type t we have the following properties:*

1. *$\sqsubseteq$ is a pre-order on $\mathcal{S}^\forall(t)$*
2. *$\sqsubseteq$ is a partial order on $\mathcal{S}^\forall(t)/\cong$*
3. *$\mathcal{S}^\forall(t)/\cong$ is finite*
4. *$(\mathcal{S}^\forall(t)/\cong, \sqsubseteq)$ is a cpo*
5. *$\bot_t$ is (a representative of) the bottom element of $\mathcal{S}^\forall(t)/\cong$. Hence $(\mathcal{S}^\forall(t)/\cong, \sqsubseteq)$ is pointed.*

[7] Namely the set of labels occurring in the analysed fix-bound expression.

Proof

1. $\sqsubseteq$ is clearly transitive

   $$\sigma \sqsubseteq \sigma' \text{ and } ; x : \sigma' \vdash^{fix} x : \sigma'' \text{ implies } ; x : \sigma \vdash^{fix} x : \sigma''$$

   by lemma 5.7, and reflexive

   $$; x : \sigma \vdash^{fix} x : \sigma$$

2. $\sqsubseteq$ is obviously well-defined and anti-symmetric on $\mathcal{S}^{\forall}(t)/\cong$.

3. Let $\forall\vec{\alpha}.C \Rightarrow \kappa$ be any type scheme in $\mathcal{S}^{\forall}(t)$. Without loss of generality we can assume that all variables in $\vec{\alpha}$ occur in $C \Rightarrow \kappa$.

   Let A be any environment with the same free flow variables as $\forall\vec{\alpha}.C \Rightarrow \kappa$. The judgement $C; A \vdash^{fix} \text{fix}\, x.x : \kappa$ is derivable:

   $$\dfrac{\dfrac{}{C; A, x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash^{fix} x : \forall\vec{\alpha}.C \Rightarrow \kappa} \qquad C \vdash C[\vec{\alpha}/\vec{\alpha}]}{C; A \vdash^{fix} \text{fix}\, x.x : \kappa}$$

   By the definition of A we have that $\forall\vec{\alpha}.C \Rightarrow \kappa = close_A(C \Rightarrow \kappa)$. From theorem 5.15 (which is clearly also applicable to this system) we find that there exists a judgement $C'; A \vdash^{fix} \text{fix}\, x.x : \kappa'$ such that $C'; A \vdash^{fix} \text{fix}\, x.x : \kappa'$ and $C; A \vdash^{fix} \text{fix}\, x.x : \kappa$ are equivalent and C' contains only flow variables α occurring free in A or κ' and for each α there is only one constraint of the form $L \subseteq \alpha$ in C'. By lemma 5.9 we then have that $close_A(C' \Rightarrow \kappa') \cong close_A(C \Rightarrow \kappa) = \forall\vec{\alpha}.C \Rightarrow \kappa$.

   It should be clear that for each t, there exist only finitely many type schemes $close_A(C' \Rightarrow \kappa') \in \mathcal{S}^{\forall}(t)$ where C' contains only flow variables α occurring free in A or κ and for each α there is only one constraint of the form $L \subseteq \alpha$ in C' (since there are only finitely many subsets L of $\mathcal{L}$).

   But since every σ is equivalent to such a type scheme we have shown that $\mathcal{S}^{\forall}(t)/\cong$ is finite.

4. Any partially ordered finite set is a cpo.

5. Obvious.

□

From this point we formally begin an induction proof of the existence of principal typings over the nesting depth of fix: in the base case we have fix-free expressions and can thus assume the existence of principal types

for all expressions. This is then used for proving that a fix-expression has a principal type. The induction step (maximal nesting depth n) does not differ from this since we by induction can assume that all expressions with nesting depth $< n$ have principal types.

We then define our version of the function $F_{A,e,x}$ defined by Mycroft.

Definition 5.30 *Define* $F_{A,e,x}(\sigma) = \sigma'$ *iff*

1. $\dfrac{\mathcal{T}}{C; A, x : \sigma \vdash_n^{fix} e : \kappa}$ *is a derivation such that* $C; A, x : \sigma \vdash_n^{fix} e : \kappa$ *is a principal typing under* A *(i.e. for every* C', κ' *s.t.* $\dfrac{\mathcal{T}'}{C'; A, x : \sigma \vdash_n^{fix} e : \kappa'}$ *we have that* $C'; A, x : \sigma \vdash_n^{fix} e : \kappa'$ *is a (halbstark) instance of* $C; A, x : \sigma \vdash_n^{fix} e : \kappa$*)*

2. $\sigma' = close_{A,x:\sigma}(C \Rightarrow \kappa)$.

If t is the type of e and x then, clearly, $F_{A,e,x}$ is a function from $\mathcal{S}^{\forall}(t)/\cong$ to $\mathcal{S}^{\forall}(t)/\cong$. It is, however, not a function from $\mathcal{S}^{\forall}(t)$ to $\mathcal{S}^{\forall}(t)$ as it allows a choice of principal typing in *1.* — we will use this freedom later to choose a principal type with "nice" properties.

Lemma 5.31 *If* $\forall\vec{\alpha}.C \Rightarrow \kappa \sqsubseteq \forall\vec{\beta}.C' \Rightarrow \kappa'$ *then every variable* α' *free in* $\forall\vec{\alpha}.C \Rightarrow \kappa$ *is also free in* $\forall\vec{\beta}.C' \Rightarrow \kappa'$.

Proof By contradiction. Assume that α' is free in $\forall\vec{\alpha}.C \Rightarrow \kappa$ but not in $\forall\vec{\beta}.C' \Rightarrow \kappa'$. If $\vec{\beta} = \langle\beta_1 \cdots \beta_n\rangle$ and $\vec{\alpha} = \langle\alpha_1 \cdots \alpha_m\rangle$ we have that α' is free in C or κ and $\alpha' \neq \alpha_i$ for all i $(1 \leq i \leq n)$ and either

1. α' not free in $C' \Rightarrow \kappa'$ or
2. $\alpha' = \beta_j$ for some j $(1 \leq j \leq m)$.

By lemma 5.6 we find that there exists S working on $\vec{\alpha}$ such that

1. $C' \vdash S(C)$,
2. $C' \vdash^{\forall} S(\kappa) \leq \kappa'$ and
3. No β_i is free in $\forall\vec{\alpha}.C \Rightarrow \kappa$

Assume α' not free in $C' \Rightarrow \kappa'$. Then if α' free in C, 1. can only hold if $\alpha' \in \mathrm{Dom}(S)$ contradicting the assumption that S works on $\vec{\alpha}$ and $\alpha' \neq \alpha_i$ for all i. If α' free in κ then 2. can only hold if either $\alpha' \in \mathrm{Dom}(S)$ or α' free in κ', both leading to contradictions.

Assume $\alpha' = \beta_j$ for some j. This clearly contradicts 3.

□

Proposition 5.32 $F_{A,e,x}$ *is monotonic.*

Proof Let σ, σ' be any type schemes in $\mathcal{S}^{\forall}(t)$ such that $\sigma' \sqsubseteq \sigma$. Let $C; A, x : \sigma \vdash_n^{fix} e : \kappa$ be a principal typing of e under $A, x : \sigma$. By lemma 5.7 we find $C; A, x : \sigma' \vdash_n^{fix} e : \kappa$.

Further, let $C'; A, x : \sigma' \vdash_n^{fix} e : \kappa'$ be a principal typing of e under $A, x : \sigma'$. Thus $C; A, x : \sigma' \vdash_n^{fix} e : \kappa$ must be a halbstark instance of $C'; A, x : \sigma' \vdash_n^{fix} e : \kappa'$. By lemma 5.9 this implies $close_{A,x:\sigma'}(C' \Rightarrow \kappa') \sqsubseteq close_{A,x:\sigma'}(C \Rightarrow \kappa)$.

By lemma 5.31, $\sigma' \sqsubseteq \sigma$ implies that all free variables of σ' are free in σ. Hence $FV(A, x : \sigma') \subseteq FV(A, x : \sigma)$ and then $FV(C \Rightarrow \kappa) \setminus FV(A, x : \sigma) \subseteq FV(C \Rightarrow \kappa) \setminus FV(A, x : \sigma')$. Thus

$$close_{A,x:\sigma'}(C \Rightarrow \kappa) \sqsubseteq close_{A,x:\sigma}(C \Rightarrow \kappa)$$

We can sum up

$$\begin{array}{rcl} F_{A,e,x}(\sigma') & = & close_{A,x:\sigma'}(C' \Rightarrow \kappa') \\ & \sqsubseteq & close_{A,x:\sigma'}(C \Rightarrow \kappa) \\ & \sqsubseteq & close_{A,x:\sigma}(C \Rightarrow \kappa) \\ & = & F_{A,e,x}(\sigma) \end{array}$$

□

Proposition 5.33 *$F_{A,e,x}$ is continuous.*

Proof Any monotonic function over a finite domain is also continuous. □

By the standard fixed point theorem, this implies that $F_{A,e,x}$ has a least fixed point since $(\mathcal{S}^{\forall}(t)/\cong, \sqsubseteq)$ is a pointed cpo.

We extend the algorithm for ML-polymorphic flow analysis to fix-polymorphic flow analysis by replacing the case for fix by:

$$\begin{array}{ll} \text{fix}\, x : t.e' : & \text{let } \sigma_0 = \bot_t \\ & \text{let } A_0 = A, x : \sigma_0 \\ & \text{repeat for } i \geq 0 \\ & \qquad \text{let } (C_{i+1}, \kappa_{i+1}) = \mathcal{W}(A_i, e') \\ & \qquad \text{let } \sigma_{i+1} = close_A(C_{i+1} \Rightarrow \kappa_{i+1}) \\ & \qquad \text{let } A_{i+1} = A, x : \sigma_{i+1} \\ & \text{until } \sigma_{i+1} \sqsubseteq \sigma_i \\ & \text{in } (C_{i+1}, \kappa_{i+1}) \end{array}$$

The body of the loop is nothing but an algorithmic presentation of $F_{A,e'}$ and the loop just repeats until a fixed point is found.

Lemma 5.34 *If A is proper then $\mathcal{W}(A, e)$ terminates without failure.*

Proof The loop in the 'fix' case terminates since $\mathcal{S}^{\forall}(t)$ is finite.

□

We go on to prove the existence of principal typings and that these are computed by algorithm $\mathcal{W}$.

Theorem 5.35 *If $\mathcal{W}(A,e) = (C,\kappa)$ then there exists $\mathcal{T}_1$ such that*

1. $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C; A \vdash_n^{fix} e : \kappa}$ *and*
2. *for any $\mathcal{T}_2$ such that*

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{C'; A \vdash_n^{fix} e : \kappa'}$$

we have that there exists S such that

   (a) *$C'; A \vdash_n^{fix} e : \kappa'$ is a halbstark instance of $C; A \vdash_n^{fix} e : \kappa$ by S and*

   (b) *Let $\mathcal{L}$ be given and let ϕ, ϕ' be two environments mapping free variables of e to properties. Then for all $l \in \mathcal{L}$ we have that $C' \vdash S\phi(y,l) \subseteq \phi'(y,l)$ for all $y \in FV(e)$ implies*

$$C' \vdash S(\Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi')$$

Proof We just do the fix-case as the remaining cases are identical to the proof of theorem 5.11. Let $\mathcal{L}$, ϕ and ϕ' be given.

From the algorithm we find that

1. $(C_{i+1}, \kappa_{i+1}) = \mathcal{W}(A, x : close_A(C_i \Rightarrow \kappa_i), e)$,
2. $\sigma_i = close_A(C_i \Rightarrow \kappa_i)$
3. $\sigma_{i+1} = close_A(C_{i+1} \Rightarrow \kappa_{i+1})$
4. $\sigma_{i+1} \sqsubseteq \sigma_i$

By induction we find a derivation

$$\mathcal{T}_1' = \frac{\mathcal{T}_1''}{C_{i+1}; A, x : close_A(C_i \Rightarrow \kappa_i) \vdash_n^{fix} e : \kappa_{i+1}}$$

By definition we have that $\sigma_{i+1} = \mathcal{F}_{A,e,x}(\sigma_i)$ and clearly σ_{i+1} is a fixpoint, i.e. $\sigma_{i+1} \cong \sigma_i$. By lemma 5.8 this implies the existence of

$$\mathcal{T}_3' = \frac{\mathcal{T}_3''}{C_{i+1}; A, x : close_A(C_{i+1} \Rightarrow \kappa_{i+1}) \vdash_n^{fix} e : \kappa_{i+1}}$$

such that $C_{i+1} \vdash \Phi_{\mathcal{L}}(\mathcal{T}_3', l, \phi_1) = \Phi_{\mathcal{L}}(\mathcal{T}_1', l, \phi_1)$.

We realise that $\mathcal{T}_1 =$

$$\frac{\dfrac{\mathcal{T}_3''}{C_{i+1}; A, x : close_A(C_{i+1} \Rightarrow \kappa_{i+1}) \vdash_n^{fix} e : \kappa_{i+1}} \quad C_{i+1} \vdash^{\forall} \kappa_{i+1} \leq \kappa_{i+1} \quad C_{i+1} \vdash C_{i+1}}{C_{i+1}; A \vdash_n^{fix} \mathrm{fix}\, x.e : \kappa_{i+1}}$$

exists.

Let $\mathcal{T}_2$ be a given typing of $\mathrm{fix}\, x.e$ under A. By the syntax directed rule for fix we see that

$$\mathcal{T}_2 = \frac{\mathcal{T}_2' \quad C_2' \vdash^{\forall} \kappa_2'' \leq \kappa_2' \quad C' \vdash C_2'[\vec{\ell'}/\vec{\beta}]}{C'; A \vdash_n^{fix} \mathrm{fix}\, x.e : \kappa_2'[\vec{\ell'}/\vec{\beta}]}$$

where

$$\mathcal{T}_2' = \frac{\mathcal{T}_2''}{C_2'; A, x : \forall \vec{\beta}.C_2' \Rightarrow \kappa_2' \vdash_n^{fix} e : \kappa_2''}$$

and $\kappa' = \kappa_2'[\vec{\ell'}/\vec{\beta}]$ and $\vec{\beta} \cap FV(A) = \emptyset$.

From $\vec{\beta} \cap FV(A) = \emptyset$ we find

$$close_A(C_2' \Rightarrow \kappa_2') \sqsubseteq \forall \vec{\beta}.C_2' \Rightarrow \kappa_2$$

And then by lemma 5.8 (special case where the judgements of the first assumption are equal) we find that there exists $\mathcal{T}_5$ such that

1. $\mathcal{T}_5 = \dfrac{\mathcal{T}_5'}{C_2'; A, x : close_A(C_2' \Rightarrow \kappa_2') \vdash_n^{fix} e : \kappa_5'}$
2. $C_2' \vdash^{\forall} \kappa_5' \leq \kappa_2''$
3. For all $l \in \mathcal{L}$ we have

$$C_2' \vdash \Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi'') \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_2', l, \phi'')$$

where $\phi'' = \phi'[(x, l') \mapsto \{\}]$ for all $l' \in \mathcal{L}$.

By transitivity $C_2' \vdash^{\forall} \kappa_3 \leq \kappa_2'$ and hence $close_A(C_2' \Rightarrow \kappa_3) \sqsubseteq close_A(C_2' \Rightarrow \kappa_2')$. This implies that $close_A(C_2' \Rightarrow \kappa_2')$ must be a fixed point of $F_{A,e,x}$.

By lemma 5.8 there exists $\mathcal{T}_6$ such that

1. $\mathcal{T}_6 = \dfrac{\mathcal{T}_6'}{C_2'; A, x : close_A(C_{i+1} \Rightarrow \kappa_{i+1}) \vdash_n^{fix} e : \kappa_6'}$
2. $C_2' \vdash^{\forall} \kappa_6' \leq \kappa_5'$
3. For all $l \in \mathcal{L}$ we have

$$C_2' \vdash \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_1') \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi_1')$$

for any ϕ_1' (we will define it below).

Let C_7 be a constraint set such that

1. $C_7 \vdash C_2'$
2. $C' \vdash C_7[\vec{\ell'}/\vec{\beta}]$ and
3. $C_7[\vec{\ell'}/\vec{\beta}] \vdash C'$

It should be clear that a constraint set fulfilling these points exists (one can think of taking C_2' which fulfils 1. and 2. and strengthening it to live up to 3.). We find

1. $\mathcal{T}_7 = \dfrac{\mathcal{T}_7'}{C_7; A, x : close_A(C_{i+1} \Rightarrow \kappa_{i+1}) \vdash_n^{fix} e : \kappa_6'}$
2. $C_7 \vdash^{\forall} \kappa_6' \leq \kappa_5'$
3. For all $l \in \mathcal{L}$ we have

$$C_7 \vdash \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_1') \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi_1')$$

By the induction hypothesis, we have S' such that

1. $C_7; A, x : close_A(C_{i+1} \Rightarrow \kappa_{i+1}) \vdash_n^{fix} e : \kappa_6'$ is a halbstark instance of $C_{i+1}; A, x : \sigma_i \vdash_n^{fix} e : \kappa_{i+1}$ by S' and
2. Define ϕ_1, ϕ_1' as the least solutions to

$$\begin{aligned} \phi_1 &= \phi[(x, l') \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_1', l, \phi_1)] \\ \phi_1' &= \phi'[(x, l') \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_1')] \end{aligned}$$

for all l'. Then $C_7 \vdash S'\phi_1(y, l) \subseteq \phi_1'(y, l)$ for all y implies

$$C_7 \vdash S'(\Phi_{\mathcal{L}}(\mathcal{T}_1', l, \phi_1)) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_1')$$

We find that

1. $C_7 \vdash S'C_{i+1}$
2. $A = S'A$
3. $C_7 \vdash^{\forall} S\kappa_{i+1} \leq \kappa_6'$

From above we have $C_2' \vdash^{\forall} \kappa_6' \leq \kappa_5'$, $C_2' \vdash^{\forall} \kappa_5' \leq \kappa_2''$ and $C_2' \vdash^{\forall} \kappa_2'' \leq \kappa_2'$, so we can conclude

1. $C_7 \vdash S'C_{i+1}$
2. $A = S'A$

3. $C_7 \vdash^{\forall} S\kappa_{i+1} \leq \kappa_2'$

Furthermore, it is easy to see that

1. $C' \vdash [\vec{\ell'}/\vec{\beta}]C_7$
2. $A = [\vec{\ell'}/\vec{\beta}]A$
3. $C' \vdash^{\forall} [\vec{\ell'}/\vec{\beta}]\kappa_2' \leq \kappa'$ (actually $[\vec{\ell'}/\vec{\beta}]\kappa_2' = \kappa'$)

We can conclude that $C'; A \vdash_n^{fix} \mathrm{fix}\, x.e : \kappa'$ is a halbstark instance of $C_{i+1}; A \vdash_n^{fix} \mathrm{fix}\, x.e : \kappa$ by $S = [\vec{\ell'}/\vec{\beta}] \circ S'$. This proves *(a)* of the theorem.

We need to prove that $C' \vdash S\phi(y,l) \subseteq \phi'(y,l)$ implies $C_7 \vdash S'\phi_1(y,l) \subseteq \phi_1'(y,l)$. Let y, l be given.

Without loss of generality, we can assume that no $\beta \in \vec{\beta}$ is free in $\phi'(y,l)$ and hence that $\phi'(y,l) = [\vec{\ell'}/\vec{\beta}]\phi'(y,l)$. Now $C' \vdash S\phi(y,l) \subseteq \phi'(y,l)$ implies $C_7[\vec{\ell'}/\vec{\beta}] \vdash [\vec{\ell'}/\vec{\beta}](S'\phi(y,l)) \subseteq [\vec{\ell'}/\vec{\beta}]\phi'(y,l)$, which in turn implies $C_7 \vdash S'\phi(y,l) \subseteq \phi'(y,l)$. For $y \neq x$ this implies $C_7 \vdash S'\phi_1(y,l) \subseteq \phi_1'(y,l)$, otherwise the result follows by an inductive argument on the fixed point computation of ϕ_1 and ϕ_1'.

We summarise the flow related properties:

$$\begin{array}{rcl} C_{i+1} \vdash \Phi_{\mathcal{L}}(\mathcal{T}_3', l, \phi_1) & = & \Phi_{\mathcal{L}}(\mathcal{T}_1', l, \phi_1) \\ C_7 \vdash S'(\Phi_{\mathcal{L}}(\mathcal{T}_1', l, \phi_1)) & \subseteq & \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_1') \\ C_2' \vdash \Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_1') & \subseteq & \Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi_1') \\ C_2' \vdash \Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi_1') & \subseteq & \Phi_{\mathcal{L}}(\mathcal{T}_2', l, \phi_1') \end{array}$$

By the definition of Φ:

$$\begin{array}{rcl} \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi) & = & \Phi_{\mathcal{L}}(\mathcal{T}_3', l, \phi_1) \\ \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi') & = & [\vec{\ell'}/\vec{\beta}](\Phi_{\mathcal{L}}(\mathcal{T}_2', l, \phi_1')) \end{array}$$

Furthermore, $C' \vdash C_2'[\vec{\ell'}/\vec{\beta}]$, $C' \vdash C_7[\vec{\ell'}/\vec{\beta}]$ and $C_2' \vdash S'(C_{i+1})$. By using that in the above properties we find:

$$\begin{array}{rcl} S(\Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)) & = & ([\vec{\ell'}/\vec{\beta}] \circ S')(\Phi_{\mathcal{L}}(\mathcal{T}_3', l, \phi_1)) \\ C' \vdash ([\vec{\ell'}/\vec{\beta}] \circ S')(\Phi_{\mathcal{L}}(\mathcal{T}_3', l, \phi_1)) & = & ([\vec{\ell'}/\vec{\beta}] \circ S')(\Phi_{\mathcal{L}}(\mathcal{T}_1', l, \phi_1)) \\ C' \vdash ([\vec{\ell'}/\vec{\beta}] \circ S')(\Phi_{\mathcal{L}}(\mathcal{T}_1', l, \phi_1)) & \subseteq & [\vec{\ell'}/\vec{\beta}](\Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_1')) \\ C' \vdash [\vec{\ell'}/\vec{\beta}](\Phi_{\mathcal{L}}(\mathcal{T}_6, l, \phi_1')) & \subseteq & [\vec{\ell'}/\vec{\beta}](\Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi_1')) \\ C' \vdash [\vec{\ell'}/\vec{\beta}](\Phi_{\mathcal{L}}(\mathcal{T}_5, l, \phi_1')) & \subseteq & [\vec{\ell'}/\vec{\beta}](\Phi_{\mathcal{L}}(\mathcal{T}_2', l, \phi_1')) \\ [\vec{\ell'}/\vec{\beta}](\Phi_{\mathcal{L}}(\mathcal{T}_2', l, \phi_1')) & = & \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi') \end{array}$$

So we conclude

$$C' \vdash S\Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi')$$

proving *(b)* of the theorem.

□

5.3.4 Algorithm II: Bounding Kleene-Mycroft Sequences

If we hope for a practical algorithm, there are two essential problems with the above approach:

1. The constraint sets can potentially explode as in ML-polymorphic flow analysis.

2. It is not obvious how to check $\sigma \sqsubseteq \sigma'$ so we cannot immediately tell when we have reached a fixed point.

Consider the sequence $\sigma_i = F^i_{A,e,x}(\bot_t)$ of which we wish to find the limit. Let us call such a sequence a *Kleene-Mycroft sequence*. For each $\sigma_i = \forall\vec{\alpha}.C \Rightarrow \kappa$, all we know is that $C; A \vdash^{fix} e : \kappa$ is *a* principal type. We have with the *elim* function a way of transforming a principal type to a principal type with properties that we can use for bounding the size of each σ_i. Let us optimize the algorithm using this insight:

$$
\begin{array}{ll}
\mathrm{fix}\, x : t.e' : & \mathrm{let}\ \sigma_0 = \bot_t \\
& \mathrm{let}\ A_0 = A, x : \sigma_0 \\
& \mathrm{repeat\ for}\ i \geq 0 \\
& \quad \mathrm{let}\ (C'_{i+1}, \kappa_{i+1}) = \mathcal{W}(A_i, e') \\
& \quad \mathrm{let}\ C_{i+1} = \mathit{elim}(\mathrm{FV}(A) \cup \mathrm{FV}(\kappa_{i+1}), C'_{i+1}) \\
& \quad \mathrm{let}\ \sigma_{i+1} = \mathit{close}_A(C_{i+1} \Rightarrow \kappa_{i+1}) \\
& \quad \mathrm{let}\ A_{i+1} = A, x : \sigma_{i+1} \\
& \mathrm{until}\ \sigma_{i+1} \sqsubseteq \sigma_i \\
& \mathrm{in}\ (C_{i+1}, \sigma_{i+1})
\end{array}
$$

The sequence of σ_i's computed by this algorithm naturally has the same properties proven in the previous subsection as any other Kleene-Mycroft sequence. Without loss of generality we assume that our sequence looks like

$$
\begin{array}{rcl}
\bot_t & = & \forall\vec{\alpha}.C_0 \Rightarrow \kappa_0 \\
& \sqsubseteq & \forall\vec{\alpha}.C_1 \Rightarrow \kappa_0 \\
& \sqsubseteq & \forall\vec{\alpha}.C_2 \Rightarrow \kappa_0 \\
& \sqsubseteq & \cdots \\
& \sqsubseteq & \forall\vec{\alpha}.C_i \Rightarrow \kappa_0 \\
& \sqsubseteq & \cdots
\end{array}
$$

where C_0 is the empty set.

To see that this is so, consider the construction of σ_{i+1}. Let e' be the fix-bound expression with m occurrences of the fix-bound variable x. The constraint set C_{i+1} contains only variables free in κ_{i+1} or A. On the other hand, neither $\mathcal{W}(A_i, e')$ nor $\mathit{elim}(\mathrm{FV}(A) \cup \mathrm{FV}(\kappa_{i+1}), C'_{i+1})$ will ever instantiate any variables in κ_{i+1} (where we by instantiation mean replacing it with a constant or a variable free in A), so κ_{i+1} is no more than a renaming of κ_i. Thus by alpha-conversion of σ_{i+1} we find the above property.

Remember that $elim(V, C)$ was defined to be a constraint set C' obtained by eliminating all variables not in V according to lemma 5.13 and merging all superfluous $L \subseteq \alpha$ constraints according to lemma 5.14. We will change this definition slightly: let $elim(V, C) = C_3$ where

C_1 is obtained from C by eliminating all variables not in V according to lemma 5.13,
C_2 is the transitive closure of C_1,
C_3 results from C_2 by merging all superfluous $L \subseteq \alpha$ constraints according to lemma 5.14.

Note that lemma 5.14 retains the invariant that the constraint set is transitively closed. Taking the transitive closure naturally does not change the expressive power of a constraint set.

We are now able to prove that the constraint set must be growing:

Lemma 5.36 *For every i:*

1. *If $L \subseteq \alpha_j$ is in C_i and $L' \subseteq \alpha_j$ is in C_{i+1} then $L \subseteq L'$*
2. *If $\alpha \subseteq \alpha'$ is in C_i then $\alpha \subseteq \alpha'$ is in C_{i+1}.*

Proof Initially, assume that e' is fix-free. We see that C_1 contains only constraints generated from the syntax of e': the occurrences of x cannot contribute as C_0 is empty. We can assume that in the j'th occurrence we in each iteration of the repeat loop instantiate $\vec{\alpha}$ with the same "fresh" $\vec{\beta_j}$ in the x-case of $\mathcal{W}$. Then $C'_{i+1} = \bigcup_j C_i[\vec{\beta_j}/\vec{\alpha}] \cup C_1$.

We now prove the lemma by induction over i:

"$i = 0$": Obvious, since C_0 is the empty set.

"$i \geq 1$": We have

$$C_i = elim(\mathrm{FV}(A) \cup \mathrm{FV}(\kappa_i), \bigcup_j C_{i-1}[\vec{\beta_j}/\vec{\alpha}] \cup C_1)$$

and

$$C_{i+1} = elim(\mathrm{FV}(A) \cup \mathrm{FV}(\kappa_{i+1}), \bigcup_j C_i[\vec{\beta_j}/\vec{\alpha}] \cup C_1)$$

By the induction hypothesis, we find

$$\bigcup_j C_{i-1}[\vec{\beta_j}/\vec{\alpha}] \cup C_1 \subseteq \bigcup_j C_i[\vec{\beta_j}/\vec{\alpha}] \cup C_1$$

and since $\kappa_i = \kappa_{i+1} = \kappa_0$, the result follows.

If e' is not fix-free, the contribution form e is not constantly C_1 so we do an induction over the nesting depth of fix. Using the lemma as an induction hypothesis, we realise that the contribution from e must be growing.

□

If we let $\vec{\beta}$ be $\vec{\alpha} \cup FV(A)$ then each C_i can be divided as $\{L_{i1} \subseteq \beta_1, \cdots, L_{in} \subseteq \beta_n\} \cup C'_i$ where C'_i contains only constraints of the form $\beta_j \subseteq \beta_k$. Thus $|C'_i|$ is $\mathcal{O}(n^2)$.

Lemma 5.36 implies that if not $\sigma_{i+1} \sqsubseteq \sigma_i$ then either some $L_{ij} \subset L_{(i+1)j}$ (and not $L_{ij} = L_{(i+1)j}$) or $C'_i \subset C'_{i+1}$ (and not $C'_i = C'_{i+1}$). Each $L_{?j}$ can only grow $|\mathcal{L}_p|$ times (where p is the analysed program) and the growth of $C'_?$ is bounded by n^2 since this is the maximal number of constraints over $\vec{\alpha}$. Thus the length of each Kleene-Mycroft sequence is bounded by $\mathcal{O}(n \mid \mathcal{L}_p \mid + n^2)$. Since $|\mathcal{L}_p|$ is roughly the size of the program and n is bounded by the size of the type of the let-bound expression and thus in turn only by the size of the program, we have an upper quadratic bound on the length of each sequence.

5.3.5 Algorithm III: Avoiding Recomputation

If we carefully examine the complexity it does not bode well for implementations. For each fix-expression in the program we do $\mathcal{O}(n^2)$ iteration — the body of the fix-expression is analysed $\mathcal{O}(n^2)$ times. In principle the number of nested fix-expressions is bounded only by the size of the program so the resulting algorithm is exponential.

Consider two nested fix-expressions $\mathrm{fix}\, x : t_x.e$ and $\mathrm{fix}\, y : t_y.e'$ where $\mathrm{fix}\, y.e'$ is a subexpression of e. When we meet the outer fix, we continue analysing e under the assumption that x has type $\bot_{t_x}$. When we meet the inner fix-expression we find the limit of the associated Kleene-Mycroft sequence starting at $\bot_{t_y}$ resulting in some type σ_y. Using this we find a first approximation to the type of the outer fix, say σ_{x1}. Using this assumption for x we redo the whole thing for e — in particular, we start analysing the inner fix-expression using assumption $\bot_{t_y}$. We have to reiterate the inner fix-expression, as the more precise type for x might force a more precise type for e'.

The key idea of the accelerated algorithm is that the type computed "last time around" will always be less (in the generic instance relation) than the type under more precise assumptions. In the example above, when we encounter $\mathrm{fix}\, y : t_y.e'$ the second time we do not have to start a new iteration at $\bot_{t_y}$, type σ_y will do just as well.

This gives us a bound on the *sum* of *all* iterations of a given expression. While we before could give an upper bound on the number of iterations due to a specific fix-expression, we can now give a total upper bound. This will lead to a polynomial algorithm.

Write $f_x(y)$ for the unary function that results from fixing the first argument of the binary function f. We have the following property for continuous functions.

Lemma 5.37 *Let $f : D \times E \to E$ be a continuous function on the product cpo $D \times E$. Let $d \sqsubseteq d' \in D$. Let $g(d) = \bigsqcup_{i\in\omega} f_d^i(\bot)$. Then $g(d') = \bigsqcup_{i\in\omega} f_{d'}^i(\bot) = \bigsqcup_{i\in\omega} f_{d'}^i(g(d))$.*

This lemma lets us recompute a fixed point *incrementally* by starting the iteration at the *previous fixed point* — in the lemma above this is $g(d)$ — instead of starting the Kleene sequence all the way from the bottom element.

We then refine the case for 'fix': the full algorithm is given in figure 5.5. The algorithm requires that we store the flow type computed for $\mathrm{fix}\, x : t.e'$ with the expression $\mathrm{fix}\, x : t.e'$ itself. At the beginning this value is set to $\bot_t$.

5.3.6 Complexity

Again let n be the size of the analysed program with explicit standard types on all subexpressions.

The trick presented in the previous subsection speeds up the algorithm from an exponential worst-case behaviour to a polynomial worst-case (in the size of the typed program). In order to show this, we first consider the complexity of performing the test $\sigma_{i+1} \sqsubseteq \sigma_i$

Lemma 5.38 *If $\sigma, \sigma' \in \mathcal{S}^\forall(t)$ for some t then testing $\sigma \sqsubseteq \sigma'$ can be done in polynomial time $\mathcal{O}(n^4)$ if σ and σ' are the results of calls to elim() and close() as in the algorithm of figure 5.5.*

Proof According to lemma 5.36 all we have to do is check set inclusion $L \subseteq L'$ n times (where the size of L and L' is bounded by n) and set inclusion of $C_i' \subseteq C_{i+1}'$ where the size of C_i' and C_{i+1}' is bounded by n^2. Naive set inclusion can be done in quadratic time thus the total complexity is $\mathcal{O}(n^4)$. □

Recall lemma 5.17 stating a $\mathcal{O}(n^6)$ upper bound in $\mid C \mid$ on the computation of $elim(F, C)$.

Theorem 5.39 *"Accelerated" algorithm $\mathcal{W}$ computes a principal type of e in time $\mathcal{O}(n^8)$ in the size of e.*

Proof Let us say that, every time the line

$$\text{let } \sigma_{i+1} = close_A(C_{i+1}' \Rightarrow \kappa_{i+1})$$

in the accelerated algorithm $\mathcal{W}$ is executed there is a *tick*. Recall that the length of a Kleene-Mycroft sequence is at most $\mathcal{O}(m^2)$ where m was the size

$\mathcal{W}(A, e) =$ case e of

x :	if $A(x) = \forall\vec{\alpha}.C \Rightarrow \kappa$
	then $(C[\vec{\alpha'}/\vec{\alpha}], \kappa[\vec{\alpha'}/\vec{\alpha}])$ where $\vec{\alpha'}$ is fresh
$\lambda^l x : t.e'$:	let $\kappa \in \mathcal{K}^\forall(t)$ be a flow type
	with fresh variable annotations
	let $(C, \kappa') = \mathcal{W}((A, x : \kappa), e')$
	in $(C, \kappa \rightarrow^{\{l\}} \kappa')$
$e_1 @^l e_2$:	let $(C, \kappa'' \rightarrow^\ell \kappa) = \mathcal{W}(A, e_1)$
	let $(C', \kappa') = \mathcal{W}(A, e_2)$
	in $(C \cup C' \cup \mathit{constraints}(\kappa' \leq \kappa''), \kappa)$
$\mathrm{True}^l(\mathrm{False}^l)$:	$(\{\}, \mathrm{Bool}^{\{l\}})$
$\mathrm{if}^l\ e_1$ then $(e_2 : t)$ else $(e_3 : t)$:	let $\kappa \in \mathcal{K}^\forall(t)$ be a flow type
	with fresh variable annotations
	let $(C_1, \mathrm{Bool}^\ell) = \mathcal{W}(A, e_1)$
	let $(C_2, \kappa_2) = \mathcal{W}(A, e_2)$
	let $(C_3, \kappa_3) = \mathcal{W}(A, e_3)$
	let $C_{new} =$ $\mathit{constraints}(\kappa_2 \leq \kappa) \cup$
	$\mathit{constraints}(\kappa_3 \leq \kappa)$
	in $(C_1 \cup C_2 \cup C_3 \cup C_{new}, \kappa)$
$(e_1, e_2)^l$:	let $(C_1, \kappa_1) = \mathcal{W}(A, e_1)$
	let $(C_2, \kappa_2) = \mathcal{W}(A, e_2)$
	in $(C_1 \cup C_2, \kappa_1 \times^{\{l\}} \kappa_2)$
$\mathrm{let}^l\ (x, y)$ be e_1 in e_2 :	let $(C_1, \kappa_x \times^\ell \kappa_y) = \mathcal{W}(A, e_1)$
	let $(C_2, \kappa) = \mathcal{W}((A, x : \kappa_x, y : \kappa_y), e_2)$
	in $(C_1 \cup C_2, \kappa)$
$\mathrm{fix}\, x : t.e'$:	let $\sigma_0 =$ most recent binding-time type
	computed for $e = \mathrm{fix}\, x : t.e'$
	let $A_0 = A, x : \sigma_0$
	repeat for $i \geq 0$
	let $(C_{i+1}, \kappa_{i+1}) = \mathcal{W}(A_i, e')$
	let $C'_{i+1} = \mathit{elim}(\mathrm{FV}(A) \cup \mathrm{FV}(\kappa_{i+1}), C_{i+1})$
	let $\sigma_{i+1} = \mathit{close}_A(C'_{i+1} \Rightarrow \kappa_{i+1})$
	let $A_{i+1} = A, x : \sigma_{i+1}$
	until $\sigma_{i+1} \sqsubseteq \sigma_i$
	in (C_{i+1}, σ_{i+1})
let $x = e_1$ in e_2 :	let $(C_1, \kappa) = \mathcal{W}(A, e_1)$
	let $C'_1 = \mathit{elim}(\mathrm{FV}(A) \cup \mathrm{FV}(\kappa), C_1)$
	let $\sigma = \mathit{close}_A(C'_1 \Rightarrow \kappa)$
	let $(C_2, \kappa') = \mathcal{W}((A, x : \sigma), e_2)$
	in (C_2, κ')

Figure 5.5: Algorithm $\mathcal{W}$ for polymorphic recursion

of the standard type of the fix-expression. Then the *total* number of ticks for a given fix-expression is bounded by $\mathcal{O}(m^2)$.

The *total* number of ticks for *all* fix-expression is then bounded by $\mathcal{O}(n^2)$.

If we let n' be the complexity of testing $\sigma_{i+1} \sqsubseteq \sigma_i$ and n'' the complexity of $elim(\mathrm{FV}(A) \cup \mathrm{FV}(\kappa_{i+1}), C_{i+1})$ we have the total complexity as $\mathcal{O}(n^2(n' + n''))$. Since n' is $\mathcal{O}(n^6)$ and n'' is $\mathcal{O}(n^4)$ we get $\mathcal{O}(n^8)$.

□

While the above complexity result does not bode well for implementations, we believe that it is overly conservative and will probably be better behaved in practice. Furthermore, we believe that the algorithm given can be improved [DHM95b].

5.3.7 Soundness

We can prove strong subject reduction for the fix-polymorphic system. We will not state the substitution lemma, as it is identical to lemma 5.19.

The lemma for fix-unfolding (equivalent to lemma 5.20) looks as follows

Lemma 5.40 *Let*

$$\mathcal{T}_1 = \frac{\dfrac{\mathcal{T}_1'}{C; A, x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{fix} e : \kappa'} \qquad C \vdash^{\forall} \kappa' \leq \kappa \quad C' \vdash C[\vec{\ell}/\vec{\alpha}]}{C'; A \vdash_n^{fix} \mathrm{fix}\, x.e : \kappa[\vec{\ell}/\vec{\alpha}]}$$

and $(\vec{\alpha} \cap (\mathrm{FV}(A) \cup \mathrm{FV}(C'))) = \emptyset$. *Then there exists* $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C'; A \vdash_n^{fix} e[\mathrm{fix}\, x.e/x] : \kappa_2}$
2. $C' \vdash \kappa_2 \leq \kappa[\vec{\ell}/\vec{\alpha}]$ *and*
3. *For any* ϕ *and* $\mathcal{L}$ *and all* $l \in \mathcal{L}$ *we have*

$$C' \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)$$

Proof We first realise (by a similar induction) that lemma 5.20 also holds for the system with polymorphic recursion. This implies the existence of

$$\mathcal{T}_3 = \frac{\mathcal{T}_3'}{C; A \vdash_n^{fix} e[\mathrm{fix}\, x.e/x] : \kappa'}$$

such that for any ϕ and $\mathcal{L}$ and all $l \in \mathcal{L}$ we have

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi) = \Phi_{\mathcal{L}}(\frac{\mathcal{T}_1'}{C; A, x : \forall\vec{\alpha}.C \Rightarrow \kappa \vdash_n^{fix} e : \kappa'}, l, \phi')$$

where $\phi' = \phi[(x, l') \mapsto \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi')]$.

It is a trivial induction to show that this implies the existence of a derivation

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{C; A \vdash_n^{fix} e[\text{fix}\, x.e/x] : \kappa'[\vec{\ell}/\vec{\alpha}]}$$

such that for any ϕ and $\mathcal{L}$ and all $l \in \mathcal{L}$ we have

$$C' \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)$$

Clearly, $C' \vdash \kappa'[\vec{\ell}/\vec{\alpha}] \leq \kappa[\vec{\ell}/\vec{\alpha}]$.

□

We then prove the main subject reduction theorem which is also extended with preservation of types of subexpressions.

Theorem 5.41 (Subject Reduction) *If* $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{C; A \vdash_n^{fix} e_1 : \kappa_1}$ *and* $e_1 \longrightarrow e_2$ *then there exists* $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C; A \vdash_n^{fix} e_2 : \kappa_2}$
2. $C \vdash^{\forall} \kappa_1 \leq \kappa_2$
3. *For any* ϕ *and* $\mathcal{L}$ *and all* $l \in \mathcal{L}$ *we have*

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)$$

and if $l \in \text{Destructors}(e_1)$ *consumes* $l' \in \text{Contructors}(e_1)$ *then for any* ϕ *and* $\mathcal{L}$ *(where* $l, l' \in \mathcal{L}$*) then*

$$C \vdash \{l'\} \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_1, l, \phi)$$

Proof The non-trivial cases follow from lemmas 5.19 and 5.40. □

Again we have soundness of flow functions computed from syntax directed inference trees (soundness of flow functions computed from non-syntax directed inference trees follows by definition):

Corollary 5.42 *Let* $\mathcal{T}$ *be any derivation for* e *and let* $C; A \vdash_n^{fix} e : \kappa$ *be its conclusion. Then* $C; \mathcal{F}_{\mathcal{T}} \models e$.

5.3.8 Invariance under Transformation

We will now show that with polymorphic recursion, no improvement is obtained by unfolding recursive definitions. The similar theorem for unfolding let-definitions still holds for the extended system.

We note that lemma 5.23 remains true after the addition of polymorphic recursion.

Theorem 5.43 (Invariance under fix-unrolling) *If*

$$\mathcal{T}_{e[\text{fix}\,x.e/x]} = \frac{\mathcal{T}}{C; A \vdash_n^{fix} e[\text{fix}\,x.e/x] : \kappa}$$

then there exists $\mathcal{T}_{\text{fix}}$ *such that*

1. $\mathcal{T}_{\text{fix}} = \dfrac{\mathcal{T}'_{\text{fix}}}{C; A \vdash_n^{fix} \text{fix}\,x.e : \kappa'}$
2. $C \vdash^{\forall} \kappa' \leq \kappa$
3. *For any* ϕ *and* $\mathcal{L}$ *and all* $l \in \mathcal{L}$ *we have*

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_{\text{fix}}, l, \phi) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_{e[\text{fix}\,x.e/x]}, l, \phi)$$

Proof By lemma 5.23, we find that

1. $\mathcal{T}_i = \dfrac{\mathcal{T}'_i}{C; A \vdash_n^{fix} \text{fix}\,x.e : \kappa_i}$
2. $\mathcal{T}_e = \dfrac{\mathcal{T}'_e}{C; A, x_1 : \kappa_1, \cdots, x_n : \kappa_n \vdash_n^{fix} e[x_i/x^{(i)}] : \kappa}$ and
3. For any ϕ and $\mathcal{L}$ and all $l \in \mathcal{L}$ we have

$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_{e[\text{fix}\,x.e/x]}, l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}_e, l, \phi) \sqcup \bigsqcup_i \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi)$$

By theorem 5.35 we know that we can derive a principal typing $\mathcal{T} = \dfrac{\mathcal{T}'}{C'; A \vdash_n^{fix} \text{fix}\,x.e : \kappa'}$ for $\text{fix}\,x.e$ under A. I.e. for every i there exists S_i such that

1. $C; A \vdash_n^{fix} \text{fix}\,x.e : \kappa_i$ is a halbstark instance of $C'; A \vdash_n^{fix} \text{fix}\,x.e : \kappa'$ under S_i and
2. Let $\phi'_i = S_i\phi$. Since by definition $C \vdash S_i\phi(y, l) \subseteq \phi'(y, l)$ for all y and $l \in \mathcal{L}$ we have
$$C \vdash S_i(\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi)) \subseteq \Phi(\mathcal{T}_i, l, \phi'_i)$$

As in the proof of theorem 5.24, we can without loss of generality assume that S_i is the identity on ϕ and hence that $\phi'_i = \phi$ and further that

$$C \vdash S_i(\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi)) \subseteq \Phi(\mathcal{T}_i, l, \phi)$$

By the construction of Kleene-Mycroft sequences, this implies that $\mathcal{T}' =$

$$\frac{\mathcal{T}''}{C'; A, x : close_A(C' \Rightarrow \kappa') \vdash_n^{fix} e : \kappa'}$$

has a conclusion which is a principal typing for e under $A, x : close_A(C' \Rightarrow \kappa')$. By definition $\Phi_{\mathcal{L}}(\mathcal{T}', l, \phi) = \Phi_{\mathcal{L}}(\mathcal{T}, l, \phi)$

By repeated application of lemma 5.8 we find

1. $\mathcal{T}_2 =$

$$\frac{\mathcal{T}_2'}{C; A, x_1 : close_A(C' \Rightarrow \kappa'), \cdots, x_n : close_A(C' \Rightarrow \kappa') \vdash_n^{fix} e[x_i/x^{(i)}] : \kappa''}$$

2. $C \vdash^{\forall} \kappa'' \leq \kappa$

3. We have
$$C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_e, l, \phi)$$

From 1. we find $\mathcal{T}_3 =$

$$\frac{\mathcal{T}_3'}{C; A, x : close_A(C' \Rightarrow \kappa') \vdash_n^{fix} e : \kappa''}$$

where

1. $C; A, x : close_A(C' \Rightarrow \kappa') \vdash_n^{fix} e : \kappa''$ is a halbstark instance of $C'; A, x : close_A(C' \Rightarrow \kappa') \vdash_n^{fix} e : \kappa'$ under some S and

2. For all y and $l \in \mathcal{L}$ we have
$$C \vdash S(\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi)) \subseteq \Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi)$$

By point 1. $C \vdash SC'$ and $C \vdash^{\forall} S\kappa' \leq \kappa$
We build $\mathcal{T}_{\text{fix}} =$

$$\frac{\dfrac{\mathcal{T}''}{C'; A, x : close_A(C' \Rightarrow \kappa') \vdash_n^{fix} e : \kappa'} \quad C' \vdash^{\forall} \kappa' \leq \kappa' \quad C \vdash SC'}{C; A \vdash_n^{fix} \text{fix}\, x.e : S\kappa'}$$

Now

$$\begin{array}{rcl}
\Phi_{\mathcal{L}}(\mathcal{T}_{\text{fix}}, l, \phi) & = & S\Phi_{\mathcal{L}}(\mathcal{T}', l, \phi) \\
 & = & S\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi) \\
C \vdash S(\Phi_{\mathcal{L}}(\mathcal{T}, l, \phi)) & \subseteq & \Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi) \\
\Phi_{\mathcal{L}}(\mathcal{T}_3, l, \phi) & = & \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi) \\
C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_2, l, \phi) & \subseteq & \Phi_{\mathcal{L}}(\mathcal{T}_e, l, \phi) \\
\vdash \Phi_{\mathcal{L}}(\mathcal{T}_e, l, \phi) & \subseteq & \Phi_{\mathcal{L}}(\mathcal{T}_e, l, \phi) \sqcup \bigsqcup_i \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi) \\
C \vdash \Phi_{\mathcal{L}}(\mathcal{T}_e, l, \phi) \sqcup \bigsqcup_i \Phi_{\mathcal{L}}(\mathcal{T}_i, l, \phi) & = & \Phi_{\mathcal{L}}(\mathcal{T}_{e[\text{fix}\, x.e/x]}, l, \phi)
\end{array}$$

□

Similar to corollary 5.25 we find

Corollary 5.44 *If*

$$\mathcal{T}_1 = \frac{\mathcal{T}_1'}{C; A \vdash_n^{fix} C[e[\text{fix}\, x.e/x]] : \kappa_1}$$

then there exists $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{C; A \vdash_n^{fix} C[\text{fix}\, x.e] : \kappa_2}$
2. $C \vdash^{\forall} \kappa_2 \leq \kappa_1$
3. *For all* $l \in \text{Destructors}(C[e[\text{fix}\, x.e/x]])$ *we have*

$$C \vdash \mathcal{F}_{\mathcal{T}_2}(l) \subseteq \mathcal{F}_{\mathcal{T}_1}(l)$$

Chapter 6

Intersection Types

Intersection types allow us to state more than one property of an expression and use any of the properties at will. Intersection types are more powerful than F2 polymorphism: any polymorphic type can be regarded as an infinite intersection where each component has a certain fixed structure. In any given program only a finite number of instantiations of the polymorphic type can be used, and thus we can write the polymorphic type as a finite intersection. This relation is true for the same reason for annotated types[1].

Example 6.1 Consider the following expression:

$$\begin{array}{l}\text{let } app = \lambda^{l_1} f.\lambda^{l_2} x.f@^{l_3}x \\ \text{ in } \cdots app@^{l_4}(\lambda^{l_5} y.y)@^{l_{11}}\text{True}^{l_6} \cdots app@^{l_7}(\lambda^{l_8} z.\text{False}^{l_9})@^{l_{12}}\text{True}^{l_{10}} \cdots\end{array}$$

The point is that function *app* is applied both to an identity function and to a constant function. With intersection types we can give function *app* the following annotated type:

$$\begin{array}{ll} & ((\text{Bool}^{\{l_6\}} \to^{\{l_5\}} \text{Bool}^{\{l_6\}}) \to^{\{l_1\}} \text{Bool}^{\{l_6\}} \to^{\{l_2\}} \text{Bool}^{\{l_6\}}) \\ \wedge & ((\text{Bool}^{\{l_{10}\}} \to^{\{l_8\}} \text{Bool}^{\{l_9\}}) \to^{\{l_1\}} \text{Bool}^{\{l_{10}\}} \to^{\{l_2\}} \text{Bool}^{\{l_9\}})\end{array}$$

Thus the result of the two applications in the body of the let can be given exact descriptions. This example can be handled with let-polymorphism, but it should be clear that intersection type based flow analysis is strictly more precise than let- and fix-polymorphism[2].

□

[1] In a sense even more true: if we assume a finite set of labels and flow variables, any polymorphic type is a representation of a finite intersection.

[2] For standard types, more expressions can be typed using intersection than System F polymorphism. This is shown by exhibiting a strongly normalising lambda term which is not typable in System F [GRDR88]: let I be $\lambda x.x$, K be $\lambda x.\lambda y.x$ and Δ be $\lambda x.x@x$ then $(\lambda x.\lambda y.y@(x@I)(x@K))@\Delta$ is not typable in System F. This expression does not show that intersection based flow analysis is more precise than System F based flow analysis (see section 8.5), and it is an open question whether this is true.

The goal of this chapter is not immediately practical: the analysis will be so precise that we cannot expect the analysis to have efficient implementations (due to Statman's lemma the problem solved is non-elementary recursive). The purpose is rather to obtain a better understanding of the problem by identifying an analysis that is "exact" (in a sense that we will make precise). We hope that this understanding will prove useful for the design of future analyses, and we believe that the characterisation given is interesting in its own right.

Since the aim of this chapter is different from the aim of previous chapters, we will not discuss algorithms and principality. Consequently, we have no use for label variables and constraint sets, so we will leave these out to avoid unnecessary clutter. Hence, properties in this chapter are simply label sets L.

We will use a version of intersection types that includes a subtype ordering. This originates from work by Barendregt, Coppo and Dezani-Ciancaglini [BCDC83].

The analysis presented here resembles the strictness analysis given by Jensen [Jen92] who, using intersection types, defined a strictness analysis equivalent to Burn, Hankin and Abramsky's abstract interpretation formulation [BHA86].

Formulae:

$$\text{Bool}\ \frac{}{\text{Bool}^L \in \mathcal{K}^\wedge(\text{Bool})}$$

$$\to\ \frac{\kappa \in \mathcal{K}^\wedge(t) \quad \kappa' \in \mathcal{K}^\wedge(t')}{\kappa \to^L \kappa' \in \mathcal{K}^\wedge(t \to t')} \qquad \times\ \frac{\kappa \in \mathcal{K}^\wedge(t) \quad \kappa' \in \mathcal{K}^\wedge(t')}{\kappa \times^L \kappa' \in \mathcal{K}^\wedge(t \times t')}$$

$$\wedge\ \frac{\kappa \in \mathcal{K}^\wedge(t) \quad \kappa' \in \mathcal{K}^\wedge(t)}{\kappa \wedge \kappa' \in \mathcal{K}^\wedge(t)}$$

Figure 6.1: Intersection flow analysis — formulae

The set of formulae presented in figure 6.1 is the same as the formulae for subtyping extended with the new intersection operator $\wedge$. Note that due to our requirement that the language is well-typed under simple (standard) typing, the individual components of an intersection will have exactly the same underlying type structure.

Figure 6.2 presents the logical rules for the system. It contains four new rules on annotated types. The first two say that anything that has type $\kappa \wedge \kappa'$ can be given type κ or κ'. The third states that if κ is smaller than

Logical rules:

$$\wedge\text{-E}\ \frac{}{\vdash^{\wedge} \kappa \wedge \kappa' \leq \kappa} \qquad \frac{}{\vdash^{\wedge} \kappa \wedge \kappa' \leq \kappa'}$$

$$\wedge\ \frac{\vdash^{\wedge} \kappa \leq \kappa_1 \quad \vdash^{\wedge} \kappa \leq \kappa_2}{\vdash^{\wedge} \kappa \leq \kappa_1 \wedge \kappa_2} \qquad \text{Trans}\ \frac{\vdash^{\wedge} \kappa_1 \leq \kappa_2 \quad \vdash^{\wedge} \kappa_2 \leq \kappa_3}{\vdash^{\wedge} \kappa_1 \leq \kappa_3}$$

Figure 6.2: Intersection flow analysis — logical rules

κ_1 and smaller than κ_2 then it is also smaller than $\kappa_1 \wedge \kappa_2$. The last rule states transitivity for subtyping.

Figure 6.3 contains the usual three rules lifting the $\subseteq$ relation to a relation on annotated types. Furthermore, it contains distribution rules for $\rightarrow$ and $\times$ over intersections. Note, that these three rules introduce equivalences since the opposite subtypings are derivable. E.g.

$$\frac{\dfrac{\dfrac{}{\vdash^{\wedge} \kappa_1 \leq \kappa_1} \quad \dfrac{}{\vdash^{\wedge} \kappa_2 \wedge \kappa_2' \leq \kappa_2}}{\vdash^{\wedge} \kappa_1 \rightarrow^L \kappa_2 \wedge \kappa_2' \leq \kappa_1 \rightarrow^L \kappa_2} \quad \dfrac{\dfrac{}{\vdash^{\wedge} \kappa_1 \leq \kappa_1} \quad \dfrac{}{\vdash^{\wedge} \kappa_2 \wedge \kappa_2' \leq \kappa_2'}}{\vdash^{\wedge} \kappa_1 \rightarrow^L \kappa_2 \wedge \kappa_2' \leq \kappa_1 \rightarrow^L \kappa_2'}}{\vdash^{\wedge} \kappa_1 \rightarrow^L \kappa_2 \wedge \kappa_2' \leq (\kappa_1 \rightarrow^L \kappa_2) \wedge (\kappa_1 \rightarrow^L \kappa_2')}$$

The non-logical rules of figure 6.4 are the same as the subtyping system (except constraint sets are left out).

Figure 6.5 presents the semi logical rules for intersection types. Apart from the subtype rule, we have a rule stating that if e has got type κ and type κ' then it has got type $\kappa \wedge \kappa'$.

6.1 Interpreting Intersection Type Derivations

There is an important difference between derivations with intersection types and the derivations we have seen previously: there can be more than one judgement for a given expression in an intersection derivation. So when we wish to find the set of values an expression e can evaluate to, we have to find *all* judgements for e (using labels, we know exactly which they are, and do not have to rely on syntactical equivalence) and take the union of the flow computed by each judgement. Our definitions of flow functions, however, are already capable of handling this situation:

If $\mathcal{T}$ is derivation, then for every l we let $\mathcal{F}_{\mathcal{T}}(l)$ be the least annotation

Type-specific rules:

$$\text{Bool}\ \frac{L_1 \subseteq L_2}{\vdash^{\wedge} \text{Bool}^{L_1} \leq \text{Bool}^{L_2}}$$

$$\text{Arrow}\ \frac{\vdash^{\wedge} \kappa_1 \leq \kappa_1' \quad \vdash^{\wedge} \kappa_2 \leq \kappa_2' \quad L_1 \subseteq L_2}{\vdash^{\wedge} \kappa_1' \rightarrow^{L_1} \kappa_2 \leq \kappa_1 \rightarrow^{L_2} \kappa_2'}$$

$$\text{Product}\ \frac{\vdash^{\wedge} \kappa_1 \leq \kappa_1' \quad \vdash^{\wedge} \kappa_2 \leq \kappa_2' \quad L_1 \subseteq L_2}{\vdash^{\wedge} \kappa_1 \times^{L_1} \kappa_2 \leq \kappa_1' \times^{L_2} \kappa_2'}$$

$$\wedge\text{-arrow}\ \frac{}{\vdash^{\wedge} (\kappa_1 \rightarrow^{L} \kappa_2) \wedge (\kappa_1 \rightarrow^{L} \kappa_2') \leq \kappa_1 \rightarrow^{L} \kappa_2 \wedge \kappa_2'}$$

$$\wedge\text{-pair-L}\ \frac{}{\vdash^{\wedge} (\kappa_1 \times^{L} \kappa_2) \wedge (\kappa_1' \times^{L} \kappa_2) \leq (\kappa_1 \wedge \kappa_1') \times^{L} \kappa_2}$$

$$\wedge\text{-pair-R}\ \frac{}{\vdash^{\wedge} (\kappa_1 \times^{L} \kappa_2) \wedge (\kappa_1 \times^{L} \kappa_2') \leq \kappa_1 \times^{L} (\kappa_2 \wedge \kappa_2')}$$

Figure 6.3: Intersection flow analysis — type-specific rules

Non-logical rules:

$$\text{Id}\ \frac{}{A, x : \kappa \vdash^{\wedge} x : \kappa}$$

$$\rightarrow\text{-I}\ \frac{A, x : \kappa \vdash^{\wedge} e : \kappa'}{A \vdash^{\wedge} \lambda^{l} x.e : \kappa \rightarrow^{\{l\}} \kappa'}$$

$$\rightarrow\text{-E}\ \frac{A \vdash^{\wedge} e : \kappa' \rightarrow^{L} \kappa \quad A \vdash^{\wedge} e' : \kappa'}{A \vdash^{\wedge} e@^{l} e' : \kappa}$$

$$\text{Bool-I}\ \frac{}{A \vdash^{\wedge} \text{True}^{l} : \text{Bool}^{\{l\}}} \qquad \frac{}{A \vdash^{\wedge} \text{False}^{l} : \text{Bool}^{\{l\}}}$$

$$\text{Bool-E}\ \frac{A \vdash^{\wedge} e : \text{Bool}^{L} \quad A \vdash^{\wedge} e' : \kappa \quad A \vdash^{\wedge} e'' : \kappa}{A \vdash^{\wedge} \text{if}^{l}\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

$$\times\text{-I}\ \frac{A \vdash^{\wedge} e : \kappa \quad A \vdash^{\wedge} e' : \kappa'}{A \vdash^{\wedge} (e, e')^{l} : \kappa \times^{\{l\}} \kappa'}$$

$$\times\text{-E}\ \frac{A \vdash^{\wedge} e : \kappa \times^{L} \kappa' \quad A, x : \kappa, y : \kappa' \vdash^{\wedge} e' : \kappa''}{A \vdash^{\wedge} \text{let}^{l}\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{fix}\ \frac{A, x : \kappa \vdash^{\wedge} e : \kappa}{A \vdash^{\wedge} \text{fix}\, x.e : \kappa} \qquad \text{let}\ \frac{A \vdash^{\wedge} e : \kappa \quad A, x : \kappa \vdash^{\wedge} e' : \kappa'}{A \vdash^{\wedge} \text{let}\ x = e\ \text{in}\ e' : \kappa'}$$

Figure 6.4: Intersection flow analysis — non-logical rules

Semi-logical rules:

$$\text{Sub}\ \frac{A \vdash^{\wedge} e : \kappa \quad \vdash^{\wedge} \kappa \leq \kappa'}{A \vdash^{\wedge} e : \kappa'}$$

$$\wedge\text{-I}\ \frac{A \vdash^{\wedge} e : \kappa \quad A \vdash^{\wedge} e : \kappa'}{A \vdash^{\wedge} e : \kappa \wedge \kappa'}$$

Figure 6.5: Intersection flow analysis — semi logical rules

such that whenever on of the rules

$$\rightarrow\text{-E}\ \frac{A \vdash^{\wedge} e : \kappa' \rightarrow^{L} \kappa \quad A \vdash^{\wedge} e' : \kappa'}{A \vdash^{\wedge} e@^{l} e' : \kappa}$$

$$\times\text{-E}\ \frac{A \vdash^{\wedge} e : \kappa \times^{L} \kappa' \quad A, x : \kappa, y : \kappa' \vdash^{\wedge} e' : \kappa''}{A \vdash^{\wedge} \text{let}^{l}\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{Bool-E}\ \frac{A \vdash^{\wedge} e : \text{Bool}^{L} \quad A \vdash^{\wedge} e' : \kappa \quad A \vdash^{\wedge} e'' : \kappa}{A \vdash^{\wedge} \text{if}^{l}\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

is an inference in $\mathcal{T}$ then $L \subseteq \mathcal{F}_{\mathcal{T}}(l)$.

6.2 Decidability

We will give a syntax directed version of our inference system. Coppo, Dezani-Ciancaglini and Veneri [CDCV81] and van Bakel [vB95] have used a similar technique of integrating the the non-structural rules in the elimination rules.

The first step is to combine the two non-syntax-directed rules into one. Define the relation $A \vdash^{\wedge'} e : \kappa$ by replacing the (Sub) and ($\wedge$-I) rules by the following rule:

$$\text{Sub'}\ \frac{\forall i \in I : \quad A \vdash^{\wedge'} e : \kappa_i \quad \vdash \kappa_i \leq \kappa_i'}{A \vdash^{\wedge'} e : \bigwedge_{i \in I} \kappa_i'}$$

Lemma 6.2 *1. If we can deduce $A \vdash e : \kappa$ from $A \vdash e : \kappa_1 \cdots A \vdash e : \kappa_n$ using only rules (Sub) and ($\wedge$-I) then we can deduce the same from the same assumptions using only rule (Sub').*

2. If Sub' $\frac{\forall i \in I : \quad A \vdash^{\wedge'} e : \kappa_i \quad \vdash \kappa_i \leq \kappa_i'}{A \vdash^{\wedge'} e : \bigwedge_{i \in I} \kappa_i'}$ then $A \vdash e : \bigwedge_i \kappa_i'$ can be inferred using rules (Sub) and ($\wedge$-I) from the same assumptions.

Proof Point *1.* is trivial since (Sub) and ($\wedge$-I) are special cases of (Sub').

If $I = \{1, \ldots, n\}$ then n applications of (Sub) and $n - 1$ applications of ($\wedge$-I) suffices for point *2.*.

□

The resulting system is sound and complete w.r.t. the original system:

Proposition 6.3 *Soundness: If $\frac{\mathcal{T}}{A \vdash^{\wedge'} e : \kappa}$ is a valid derivation then there exists a valid derivation $\frac{\mathcal{T}'}{A \vdash^{\wedge} e : \kappa}$ such that $\mathcal{F}_{\frac{\mathcal{T}'}{A \vdash^{\wedge} e : \kappa}}(l) = \mathcal{F}_{\frac{\mathcal{T}}{A \vdash^{\wedge'} e : \kappa}}(l)$ for all $l \in \text{Destructors}(e)$.*

Completeness: If $\dfrac{\mathcal{T}}{A \vdash^{\wedge} e : \kappa}$ *is a valid derivation then there exists a valid derivation* $\dfrac{\mathcal{T}'}{A \vdash^{\wedge'} e : \kappa}$ *such that* $\mathcal{F}_{\frac{\mathcal{T}'}{A \vdash^{\wedge'} e : \kappa}}(l) \subseteq \mathcal{F}_{\frac{\mathcal{T}}{A \vdash^{\wedge} e : \kappa}}(l)$ *for all* $l \in$ Destructors(e).

It is trivial to see that we never need two consecutive applications of rule (Sub').

Define function $normalise : \mathcal{K}(t) \to \mathcal{K}(t)$ for all t as follows: $normalise(\kappa)$ is the result of exhaustively applying the following rewrite rules to κ:

$$\begin{array}{rcll} \kappa_1 \to^L (\kappa_2 \wedge \kappa_2') & \longrightarrow & (\kappa_1 \to^L \kappa_2) \wedge (\kappa_1 \to^L \kappa_2') & \\ (\kappa_1 \wedge \kappa_1') \times^L \kappa_2 & \longrightarrow & (\kappa_1 \times^L \kappa_2) \wedge (\kappa_1' \times^L \kappa_2) & \\ \kappa_1 \times^L (\kappa_2 \wedge \kappa_2') & \longrightarrow & (\kappa_1 \times^L \kappa_2) \wedge (\kappa_1 \times^L \kappa_2') & \\ K[\kappa] & \longrightarrow & K[\kappa'] & \text{if } \kappa \longrightarrow \kappa' \end{array}$$

where type contexts K are defined by

$$K ::= [\,] \mid \kappa \times^L K \mid K \times^L \kappa \mid \kappa \to^L K$$

Note that $\vdash \kappa = normalise(\kappa)$ for all κ and that the only conjunction in "Rank 1 position" in $normalise(\kappa)$ is at top level.

A syntax directed version of the inference system is given in figure 6.6. We need a version of the property of strengthened assumptions:

Lemma 6.4 *If* $\dfrac{\mathcal{T}}{A, x : \kappa_1 \vdash_n^{\wedge} e : \kappa}$ *is a valid derivation and* $\vdash \kappa_2 \leq \kappa_1$ *then there exists* $\mathcal{T}', \kappa'$ *such that* $\dfrac{\mathcal{T}'}{A, x : \kappa_2 \vdash_n^{\wedge} e : \kappa'}$ *is a valid derivation and* $\mathcal{F}_{\frac{\mathcal{T}'}{A, x : \kappa_2 \vdash_n^{\wedge} e : \kappa'}}(l) \subseteq \mathcal{F}_{\frac{\mathcal{T}}{A, x : \kappa_1 \vdash_n^{\wedge} e : \kappa}}(l)$ *for all* $l \in$ Destructors(e).

Proof By induction over the structure of e.

□

The syntax directed system is sound and complete w.r.t. the original system in the following sense:

Theorem 6.5 *Soundness: If* $\dfrac{\mathcal{T}}{A \vdash_n^{\wedge} e : \kappa}$ *is a valid normalised derivation then there exists a valid derivation* $\dfrac{\mathcal{T}'}{A' \vdash^{\wedge} e : \kappa'}$ *such that* $\mathcal{F}_{\frac{\mathcal{T}'}{A' \vdash^{\wedge} e : \kappa'}}(l) = \mathcal{F}_{\frac{\mathcal{T}}{A \vdash_n^{\wedge} e : \kappa}}(l)$ *for all* $l \in$ Destructors(e).

Completeness: If $\dfrac{\mathcal{T}}{A \vdash^{\wedge} e : \kappa}$ *is a valid derivation then there exists a valid normalised derivation* $\dfrac{\mathcal{T}'}{A' \vdash_n^{\wedge} e : \kappa'}$ *such that* $\mathcal{F}_{\frac{\mathcal{T}'}{A' \vdash_n^{\wedge} e : \kappa'}}(l) \subseteq \mathcal{F}_{\frac{\mathcal{T}}{A \vdash^{\wedge} e : \kappa}}(l)$ *for all* $l \in$ Destructors(e).

$$\text{Id}\ \frac{}{A, x : \kappa \vdash^{\wedge}_{n} x : \kappa} \qquad \rightarrow\text{-I}\ \frac{A, x : \kappa \vdash^{\wedge}_{n} e : \kappa'}{A \vdash^{\wedge}_{n} \lambda^{l} x.e : \kappa \rightarrow^{\{l\}} \kappa'}$$

$$\rightarrow\text{-E}\ \frac{A \vdash^{\wedge}_{n} e : \kappa' \rightarrow^{L} \kappa \quad \forall i \in I : A \vdash^{\wedge}_{n} e' : \kappa'_i \quad \vdash^{\wedge} \bigwedge_{i \in I} \kappa'_i \leq \kappa'}{A \vdash^{\wedge}_{n} e @^{l} e' : \kappa}$$

$$\times\text{-I}\ \frac{A \vdash^{\wedge}_{n} e : \kappa \quad A \vdash^{\wedge}_{n} e' : \kappa'}{A \vdash^{\wedge}_{n} (e, e')^{l} : \kappa \times^{\{l\}} \kappa'}$$

$$\times\text{-E}\ \frac{\forall i \in I : A \vdash^{\wedge}_{n} e : \kappa_i \times^{L} \kappa'_i \quad A, x : \bigwedge_{i \in I} \kappa_i, y : \bigwedge_{i \in I} \kappa'_i \vdash^{\wedge}_{n} e' : \kappa''}{A \vdash^{\wedge}_{n} \text{let}^{l}\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa''}$$

$$\text{Bool-I}\ \frac{}{A \vdash^{\wedge}_{n} \text{True}^{l} : \text{Bool}^{\{l\}}} \qquad \frac{}{A \vdash^{\wedge}_{n} \text{False}^{l} : \text{Bool}^{\{l\}}}$$

$$\text{Bool-E}\ \frac{A \vdash^{\wedge}_{n} e : \text{Bool}^{L} \quad A \vdash^{\wedge}_{n} e' : \kappa' \quad A \vdash^{\wedge}_{n} e'' : \kappa'' \quad \vdash^{\wedge} \kappa' \leq \kappa \quad \vdash^{\wedge} \kappa'' \leq \kappa}{A \vdash^{\wedge}_{n} \text{if}^{l}\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

$$\text{let}\ \frac{\forall i : A \vdash^{\wedge}_{n} e : \kappa_i \quad A, x : \bigwedge_{i \in I} \kappa_i \vdash^{\wedge}_{n} e' : \kappa'}{A \vdash^{\wedge}_{n} \text{let}\ x = e\ \text{in}\ e' : \kappa'}$$

$$\text{fix}\ \frac{\forall i : A, x : \kappa \vdash^{\wedge}_{n} e : \kappa_i \quad \vdash^{\wedge} \bigwedge_{i \in I} \kappa_i \leq \kappa}{A \vdash^{\wedge}_{n} \text{fix}^{l} x.e : \kappa_j} \quad \text{for any } j \in I$$

Figure 6.6: Syntax Directed Intersection Flow Analysis

Proof We prove the theorem for $\vdash^{\wedge'}$ instead of $\vdash^{\wedge}$. By proposition 6.3 the theorem follows.

Soundness is a trivial induction since we have have just incorporated the non syntax directed rules in the syntax directed ones.

For completeness, we prove in addition to the above that

> If $\dfrac{\mathcal{T}}{A \vdash^{\wedge'} e : \kappa}$ is a valid derivation and $normalise(\kappa) = \bigwedge_{i \in I} \kappa_i$ then there exists a family $\dfrac{\mathcal{T}_i}{A \vdash^{\wedge}_n e : \kappa'_i}$ for $i \in I$ of derivations such that $\vdash^{\wedge} \kappa'_i \leq \kappa_i$ for all i.

We prove completeness by induction over the inference tree for $\dfrac{\mathcal{T}}{A \vdash^{\wedge'} e : \kappa}$:

(Id) Trivial

($\rightarrow$-I) Assume a derivation

$$\rightarrow\text{-I}\ \frac{\dfrac{\mathcal{T}}{A, x : \kappa \vdash^{\wedge'} e : \kappa'}}{A \vdash^{\wedge'} \lambda^l x.e : \kappa \rightarrow^{\{l\}} \kappa'}$$

By induction we find a family of derivation for $i \in I$:

$$\frac{\mathcal{T}_i}{A, x : \kappa \vdash^{\wedge}_n e : \kappa'_i}$$

such that if $normalise(\kappa') = \bigwedge_{i \in I} \kappa_i$ then $\vdash^{\wedge} \kappa'_i \leq \kappa_i$ for all $i \in I$. We construct

$$\rightarrow\text{-I}\ \frac{\dfrac{\mathcal{T}_i}{A, x : \kappa \vdash^{\wedge}_n e : \kappa'_i}}{A \vdash^{\wedge}_n \lambda^l x.e : \kappa \rightarrow^{\{l\}} \kappa'_i}$$

Now we have $normalise(\kappa \rightarrow^{\{l\}} \kappa') = \bigwedge_{i \in I}(\kappa \rightarrow^{\{l\}} \kappa_i)$ and

$$\vdash^{\wedge} \kappa \rightarrow^{\{l\}} \kappa'_i \leq \kappa \rightarrow^{\{l\}} \kappa_i$$

for all $i \in I$

($\rightarrow$-E) Assume

$$\rightarrow\text{-E}\ \frac{\dfrac{\mathcal{T}}{A \vdash^{\wedge'} e : \kappa' \rightarrow^{L} \kappa} \quad \dfrac{\mathcal{T}'}{A \vdash^{\wedge'} e' : \kappa'}}{A \vdash^{\wedge'} e @^l e' : \kappa}$$

Further, assume

$$\begin{aligned} normalise(\kappa' \rightarrow^{L} \kappa) &= \bigwedge_{i \in I}(\kappa' \rightarrow^{L} \kappa_i) \\ normalise(\kappa') &= \bigwedge_{j \in J} \kappa'_j \end{aligned}$$

then by induction there exists families of derivations

$$\frac{\mathcal{T}_i}{A \vdash^\wedge_n e : \kappa'_i \to^{L_i} \kappa''_i} \qquad \text{and} \qquad \frac{\mathcal{T}'_j}{A \vdash^\wedge_n e' : \kappa'''_j}$$

for $i \in I$ and $j \in J$ such that

$$\vdash^\wedge \kappa'_i \to^{L_i} \kappa''_i \leq \kappa' \to^{L} \kappa_i \qquad \text{and} \qquad \vdash^\wedge \kappa'''_j \leq \kappa'_j$$

We construct

$$\frac{\dfrac{\mathcal{T}_i}{A \vdash^\wedge_n e : \kappa'_i \to^{L_i} \kappa''_i} \quad \forall j \in J : \dfrac{\mathcal{T}'_j}{A \vdash^\wedge_n e' : \kappa'''_j} \quad \vdash^\wedge \bigwedge_{j \in J} \kappa'''_j \leq \kappa'_i}{A \vdash^\wedge_n e @^l e' : \kappa''_i}$$

where $\vdash^\wedge \bigwedge_{j\in J} \kappa'''_j \leq \kappa'_i$ follows from $\vdash^\wedge \kappa'''_j \leq \kappa'_j$, $\vdash^\wedge \bigwedge_{j\in J} \kappa'_j = \kappa'$ and $\vdash^\wedge \kappa' \leq \kappa'_i$.

We have that $\mathit{normalise}(\kappa) = \bigwedge_{i\in I} \kappa_i$ and that $\vdash^\wedge \kappa''_i \leq \kappa_i$.

($\times$-I) Assume

$$\times\text{-I}\ \frac{\dfrac{\mathcal{T}}{A \vdash^{\wedge'} e : \kappa} \quad \dfrac{\mathcal{T}'}{A \vdash^{\wedge'} e' : \kappa'}}{A \vdash^{\wedge'} (e, e')^l : \kappa \times^{\{l\}} \kappa'}$$

Further, assume

$$\begin{array}{rcl} \mathit{normalise}(\kappa) & = & \bigwedge_{i\in I}(\kappa_i) \\ \mathit{normalise}(\kappa') & = & \bigwedge_{j\in J} \kappa'_j \end{array}$$

then by induction there exists families of derivations

$$\frac{\mathcal{T}_i}{A \vdash^\wedge_n e : \kappa''_i} \qquad \text{and} \qquad \frac{\mathcal{T}'_j}{A \vdash^\wedge_n e' : \kappa'''_j}$$

for $i \in I$ and $j \in J$ such that

$$\vdash^\wedge \kappa''_i \leq \kappa_i \qquad \text{and} \qquad \vdash^\wedge \kappa'''_j \leq \kappa'_j$$

Now $\mathit{normalise}(\kappa \times^{\{l\}} \kappa') = \bigwedge_{i\in I, j\in J}(\kappa_i \times^{\{l\}} \kappa'_j)$ and for each $(i, j) \in I \times J$ we have

$$\frac{\dfrac{\mathcal{T}_i}{A \vdash^\wedge_n e : \kappa''_i} \quad \dfrac{\mathcal{T}'_j}{A \vdash^\wedge_n e' : \kappa'''_j}}{A \vdash^\wedge_n (e, e')^l : \kappa''_i \times^{\{l\}} \kappa'''_j}$$

and clearly $\vdash^\wedge \kappa''_i \times^{\{l\}} \kappa'''_j \leq \kappa_i \times^{\{l\}} \kappa'_j$.

($\times$-E) Assume

$$\times\text{-E}\ \frac{\dfrac{\mathcal{T}}{A \vdash^{\wedge'} e : \kappa_x \times^L \kappa_y} \qquad \dfrac{\mathcal{T}'}{A, x : \kappa_x, y : \kappa_y \vdash^{\wedge'} e' : \kappa}}{A \vdash^{\wedge'} \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa}$$

Further, assume

$$\begin{aligned} \mathit{normalise}(\kappa_x \times^L \kappa_y) &= \textstyle\bigwedge_{i \in I} (\kappa_{ix} \times^L \kappa_{iy}) \\ \mathit{normalise}(\kappa) &= \textstyle\bigwedge_{j \in J} \kappa_j \end{aligned}$$

then by induction there exists families of derivations

$$\frac{\mathcal{T}_i}{A \vdash^{\wedge}_n e : \kappa'_{ix} \times^{L_i} \kappa'_{iy}} \qquad \text{and} \qquad \frac{\mathcal{T}'_j}{A, x : \kappa_x, y : \kappa_y \vdash^{\wedge}_n e' : \kappa'_j}$$

for $i \in I$ and $j \in J$ such that

$$\vdash^{\wedge} \kappa'_{ix} \times^{L_i} \kappa'_{iy} \leq \kappa_{ix} \times^L \kappa_{iy} \qquad \text{and} \qquad \vdash^{\wedge} \kappa'_j \leq \kappa_j$$

It follows that $\vdash^{\wedge} \kappa'_{ix} \leq \kappa_{ix}$ and $\vdash^{\wedge} \kappa'_{iy} \leq \kappa_{iy}$ and hence

$$\frac{\dfrac{\mathcal{T}_i}{A \vdash^{\wedge}_n e : \kappa'_{ix} \times^{L_i} \kappa'_{iy}} \qquad \dfrac{\mathcal{T}'_j}{A, x : \bigwedge_{i \in I} \kappa'_{ix}, y : \bigwedge_{i \in I} \kappa'_{iy} \vdash^{\wedge}_n e' : \kappa'_j}}{A \vdash^{\wedge}_n \text{let}^l\ (x, y)\ \text{be}\ e\ \text{in}\ e' : \kappa'_j}$$

for $j \in J$ is the wanted family of derivations. Since $\vdash^{\wedge} \bigwedge_{i \in I} \kappa'_{ix} \leq \kappa_x$ and $\vdash^{\wedge} \bigwedge_{i \in I} \kappa'_{iy} \leq \kappa_y$, the use of strengthened assumptions in $\mathcal{T}'_j$ is justified by lemma 6.4.

(Bool-I) Trivial.

(Bool-E) Assume

$$\text{Bool-E}\ \frac{\dfrac{\mathcal{T}}{A \vdash^{\wedge'} e : \text{Bool}^L} \qquad \dfrac{\mathcal{T}}{A \vdash^{\wedge'} e' : \kappa} \qquad \dfrac{\mathcal{T}}{A \vdash^{\wedge'} e'' : \kappa}}{A \vdash^{\wedge'} \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

Further, assume

$$\begin{aligned} \mathit{normalise}(\text{Bool}^L) &= \text{Bool}^L \\ \mathit{normalise}(\kappa) &= \textstyle\bigwedge_{i \in I} \kappa_i \end{aligned}$$

then by induction there exists families of derivations

$$\frac{\mathcal{T}'''}{A \vdash^{\wedge}_n e : \text{Bool}^L} \qquad \frac{\mathcal{T}_i}{A \vdash^{\wedge}_n e' : \kappa'_i} \qquad \text{and} \qquad \frac{\mathcal{T}'_i}{A \vdash^{\wedge}_n e'' : \kappa''_i}$$

for $i \in I$ and $j \in J$ such that

$$\vdash^{\wedge} \kappa_i' \leq \kappa_i \qquad \text{and} \qquad \vdash^{\wedge} \kappa_i'' \leq \kappa_i$$

Now construct

$$\frac{\mathcal{T}''' \quad \overline{A \vdash_n^{\wedge} e : \mathrm{Bool}^L} \quad \mathcal{T}_i \quad \overline{A \vdash_n^{\wedge} e' : \kappa_i'} \quad \mathcal{T}_i' \quad \overline{A \vdash_n^{\wedge} e'' : \kappa_i''} \quad \vdash \kappa_i' \leq \kappa_i \quad \vdash \kappa_i'' \leq \kappa_i}{A \vdash_n^{\wedge} \mathrm{if}^l\ e \text{ then } e' \text{ else } e'' : \kappa_i}$$

(let) Assume

$$\text{let } \frac{\dfrac{\mathcal{T}}{A \vdash^{\wedge'} e : \kappa_x} \quad \dfrac{\mathcal{T}'}{A, x : \kappa_x \vdash^{\wedge'} e' : \kappa}}{A \vdash^{\wedge'} \mathrm{let}^l\ x = e \text{ in } e' : \kappa}$$

Further, assume

$$\begin{array}{rcl} \mathit{normalise}(\kappa) & = & \bigwedge_{i \in I} \kappa_{ix} \\ \mathit{normalise}(\kappa') & = & \bigwedge_{j \in J} \kappa_j \end{array}$$

then by induction there exists families of derivations

$$\frac{\mathcal{T}_i}{A \vdash_n^{\wedge} e : \kappa_{ix}'} \qquad \text{and} \qquad \frac{\mathcal{T}_j'}{A, x : \kappa_x \vdash_n^{\wedge} e' : \kappa_j'}$$

for $i \in I$ and $j \in J$ such that

$$\vdash^{\wedge} \kappa_{ix}' \leq \kappa_{ix} \qquad \text{and} \qquad \vdash^{\wedge} \kappa_j' \leq \kappa_j$$

It follows that

$$\frac{\dfrac{\mathcal{T}_i}{A \vdash_n^{\wedge} e : \kappa_{ix}'} \quad \dfrac{\mathcal{T}_j'}{A, x : \bigwedge\limits_{i \in I} \kappa_{ix}' \vdash_n^{\wedge} e' : \kappa_j'}}{A \vdash_n^{\wedge} \mathrm{let}^l\ x = e \text{ in } e' : \kappa_j'}$$

for $j \in J$ is the wanted family of derivations. Since $\vdash^{\wedge} \bigwedge_{i \in I} \kappa_{ix}' \leq \kappa_x$, the use of strengthened assumptions in $\mathcal{T}_j'$ is justified by lemma 6.4.

(fix) Assume

$$\text{fix } \frac{\dfrac{\mathcal{T}}{A, x : \kappa \vdash^{\wedge'} e : \kappa}}{A \vdash^{\wedge'} \mathrm{fix}^l x.e : \kappa}$$

Assume that $\mathit{normalise}(\kappa) = \bigwedge_{i \in I} \kappa_i$. Then by induction there exists a family of derivations

$$\frac{\mathcal{T}_i}{A, x : \kappa \vdash_n^{\wedge} e : \kappa_i'}$$

such that $\vdash^{\wedge} \kappa_i' \leq \kappa_i$ for all i. It follows that $\vdash^{\wedge} \bigwedge_{i\in I} \kappa_i' \leq \kappa$ and hence we have the following family of derivations:

$$\text{fix}\ \frac{\forall i \in I\ \dfrac{\mathcal{T}_i}{A, x : \kappa \vdash^{\wedge'} e : \kappa_i'} \qquad \vdash^{\wedge} \bigwedge_{i\in I} \kappa_i' \leq \kappa}{A \vdash^{\wedge'} \text{fix}^l x.e : \kappa_j}$$

for $j \in I$.

(Sub') Assume

$$\frac{\forall i \in I : \quad \dfrac{\mathcal{T}_i}{A \vdash^{\wedge'} e : \kappa_i} \qquad \vdash^{\wedge} \kappa_i \leq \kappa_i'}{A \vdash^{\wedge'} e : \bigwedge_{i\in I} \kappa_i'}$$

Let $\bigwedge_{j\in J_i} \kappa_{ij} = \mathit{normalise}(\kappa_i)$ for all $i \in I$. For each $i \in I$ we have by induction families of derivations

$$\frac{\mathcal{T}_{ij}}{A \vdash_n^{\wedge} e : \kappa_{ij}'}$$

where $\vdash^{\wedge} \kappa_{ij}' \leq \kappa_{ij}$ for all $i \in I$ and $j \in J_i$. Now, the family indexed by $I \times J$ fulfils the property we wish to prove.

□

We can now argue decidability as follows: first note that the subtype relation $\vdash^{\wedge} \kappa \leq \kappa'$ is decidable. Now given an expression e, the height of the normalised inference tree (leaving out the $\vdash^{\wedge} \kappa \leq \kappa'$ judgements) is bounded by the size of e. W.r.t. width, the only interesting rules are ($\rightarrow$-E) and (fix) since the number of assumptions in these rules is not fixed:

1. In the ($\rightarrow$-E) rule we have assumptions $A \vdash_n^{\wedge} e' : \kappa_i'$, but the number of such assumptions is bounded by the size of $\mathcal{K}(t)/=$ where $t = |\kappa_i'|$.

2. Similarly, in the (fix) rule we have that the number of assumptions $A, x : \kappa \vdash_n^{\wedge} e' : \kappa_i$ is bounded by the size of $\mathcal{K}(t)/=$ where $t = |\kappa_i|$.

6.3 Minimality

Define a *vectorising* function *vec* on properties as follows

$$\begin{array}{rcl} \mathit{vec}(\text{Bool}^L) & = & \langle L \rangle \\ \mathit{vec}(\kappa \times^L \kappa') & = & \mathit{vec}(\kappa) +\!\!+ \langle L \rangle +\!\!+ \mathit{vec}(\kappa') \\ \mathit{vec}(\kappa \rightarrow^L \kappa') & = & \mathit{vec}(\kappa) +\!\!+ \langle L \rangle +\!\!+ \mathit{vec}(\kappa') \\ \mathit{vec}(\kappa \wedge \kappa') & = & \mathit{vec}(\kappa) \cap \mathit{vec}(\kappa') \end{array}$$

where $\cap$ is pointwise set intersection and $+\!\!+$ is vector concatenation. Define an ordering $\preceq$ on properties

$$\kappa \preceq \kappa' \quad \text{iff} \quad vec(\kappa) \subseteq vec(\kappa')$$

(where $\subseteq$ is pointwise subset inclusion). This is extended to judgements by defining $(x_1 : \kappa_1, \cdots, x_n : \kappa_n \vdash^\wedge e : \kappa) \preceq (x_1 : \kappa'_1, \cdots, x_n : \kappa'_n \vdash^\wedge e : \kappa')$ iff $\kappa \preceq \kappa'$ and $\kappa_i \preceq \kappa'_i$ for $i \in \{1, \cdots, n\}$. We say $\mathcal{T} \preceq \mathcal{T}'$ if $\mathcal{T}$ and $\mathcal{T}'$ are derivations for the same expression e, and for all subexpressions e' of e we have that for the last (closest to the conclusion) judgements $A \vdash^\wedge e' : \kappa$ and $A' \vdash^\wedge e' : \kappa'$ (in $\mathcal{T}$ resp. $\mathcal{T}'$) it holds that $(A \vdash^\wedge e' : \kappa) \preceq (A' \vdash^\wedge e' : \kappa')$.

Define $\kappa \sqcap \kappa'$ as follows

$$\begin{array}{rcl}
\text{Bool}^{L_1} \sqcap \text{Bool}^{L_2} & = & \text{Bool}^{L_1 \cap L_2} \\
\kappa_1 \times^{L_1} \kappa'_1 \sqcap \kappa_2 \times^{L_2} \kappa'_2 & = & (\kappa_1 \sqcap \kappa_2) \times^{L_1 \cap L_2} (\kappa'_1 \sqcap \kappa'_2) \\
\kappa_1 \to^{L_1} \kappa'_1 \sqcap \kappa_2 \to^{L_2} \kappa'_2 & = & (\kappa_1 \sqcap \kappa_2) \to^{L_1 \cap L_2} (\kappa'_1 \sqcap \kappa'_2) \\
(\kappa_1 \wedge \kappa'_1) \sqcap \kappa_2 & = & (\kappa_1 \sqcap \kappa_2) \wedge (\kappa'_1 \sqcap \kappa_2) \\
\kappa_1 \sqcap \kappa_2 & = & \kappa_2 \sqcap \kappa_1
\end{array}$$

It is easy to check that $\sqcap$ is the greatest lower bound operator on the domain $(\mathcal{K}/=, \preceq)$. Define

$$\begin{array}{l}
(x_1 : \kappa_1, \cdots, x_n : \kappa_n \vdash^\wedge e : \kappa) \sqcap (x_1 : \kappa'_1, \cdots, x_n : \kappa'_n \vdash^\wedge e : \kappa') \\
\quad = (x_1 : \kappa_1 \sqcap \kappa'_1, \cdots, x_n : \kappa_n \sqcap \kappa'_n \vdash^\wedge e : \kappa \sqcap \kappa')
\end{array}$$

and for two derivations $\mathcal{T}$ and $\mathcal{T}'$ for the same expression e let $\mathcal{T} \sqcap \mathcal{T}'$ be a derivation such that the last judgement for any subexpression e' is the $\sqcap$ of the similar judgements in $\mathcal{T}$ and $\mathcal{T}'$. The following lemma shows that such a derivation exists:

Lemma 6.6 *If*

$$\frac{\mathcal{T}}{A \vdash^\wedge e : \kappa} \qquad \textit{and} \qquad \frac{\mathcal{T}'}{A' \vdash^\wedge e : \kappa'}$$

are derivations then so is

$$\frac{\mathcal{T} \sqcap \mathcal{T}'}{A \sqcap A' \vdash^\wedge e : \kappa \sqcap \kappa'}$$

Proof First prove that $\vdash^\wedge \kappa_1 \leq \kappa_2$ implies $\vdash^\wedge \kappa_1 \sqcap \kappa_3 \leq \kappa_2 \sqcap \kappa_3$ for all κ_1, κ_2 and κ_3. This follows by induction on the derivation of $\vdash^\wedge \kappa_1 \leq \kappa_2$. Now the lemma follows by induction over the sum of the heights of the two derivations.

□

It follows as an immediate consequence that

Corollary 6.7 *For each e there exists a* minimal *derivation under the $\preceq$ ordering.*

If $\mathcal{T}$ is minimal for e then $\mathcal{F}_\mathcal{T}$ is the minimal flow relation derivable for e.

6.4 Non-Standard Semantics

In this section we give a non-standard semantics which exactly characterises the strength of intersection flow analysis. The semantics is a modification of the standard semantics such that if the flow analysis predicts a potential redex, this redex will be reduced by the semantics.

Intuitively, the intersection based analysis loses information whenever computation is discarded. E.g. analysing

$$\text{if}^{l_1}\ \text{True}^{l_2}\ \text{then False}^{l_3}\ \text{else}\ (\lambda^{l_4}x.x)@^{l_5}\text{False}^{l_6}$$

will tell us that λ^{l_4} can be applied at application $@^{l_5}$. If we choose to reduce the conditional, we will discard the else-branch and therefore never perform the reduction predicted by the analysis.

We introduce new syntactic constructs to ensure that this never happens. To avoid discarding computation when 'if' is reduced, we introduce a new construct 'either': we will reduce 'if True then e else e'' to a special expression 'either e or e''. The type rule for 'either' is

$$\text{Either}\ \frac{A \vdash e : t \quad A \vdash e' : t}{A \vdash \text{either } e \text{ or } e' : t}$$

An 'either' expression cannot be reduced. The rule for flow analysis is also straightforward:

$$\text{Either}\ \frac{A \vdash^{\wedge} e : \kappa \quad A \vdash^{\wedge} e' : \kappa}{A \vdash^{\wedge} \text{either}^{l}\ e \text{ or } e' : \kappa}$$

Reduction of conditionals will thus result in a *new* expression, not present in the redex. We therefore define that the label of the conditional is taken over by the reduct, i.e. we reduce 'if^l True then e else e'' to 'either^l e or e''.

The analysis also loses information in a less obvious way: by always analysing all subexpressions of any expression the analysis assumes that no redex is ever discarded. Here redex has to be understood in a broad sense, consider:

$$\begin{array}{l}\text{let } f = \lambda x.x \\ \text{in}\ \ \text{let } g = \lambda y.f@\text{True} \\ \qquad \text{in } f@\text{False}\end{array}$$

Neither call-by-value nor call-by-name will ever meet the redex $(\lambda x.x)@\text{True}$. Thus if the above is allowed to reduce to

$$\begin{array}{l}\text{let } f = \lambda x.x \\ \text{in } f@\text{False}\end{array}$$

we cannot have subject expansion in a strong sense (that is preservation of flow). Neither do we have standard subject expansion (on judgements, not derivations) since we can derive

$$; f : \text{Bool}^{\{l_1\}} \rightarrow^{\{l_5\}} \text{Bool}^{\{l_1\}} \vdash^{\wedge} f@^{l_4}\text{False}^{l_1} : \text{Bool}^{\{l_1\}}$$

but not

$$; f : \mathrm{Bool}^{\{l_1\}} \to^{\{l_5\}} \mathrm{Bool}^{\{l_1\}} \vdash^{\wedge} \text{let } g = \lambda y.f@^{l_3}\mathrm{True}^{l_2} \text{ in } f@^{l_4}\mathrm{False}^{l_1} : \mathrm{Bool}^{\{l_1\}}$$

There are two ways out of this. Either we extend the type system with a special type Ω as known from intersection typing, but this would interfere with soundness under arbitrary reduction (arbitrary reduction *can* reduce $(\lambda x.x)@\mathrm{True}$ above) so this would restrict the applicability of the analysis — we will return to this idea in section 8.3 where we will also consider better ways of dealing with conditionals.

The other way is to define the reduction system such that no expression is *ever* discarded. For this purpose we introduce a special syntactic construct "discard e in e'". The type rule for this construct is

$$\text{Discard } \frac{A \vdash e' : \kappa' \quad A \vdash e : \kappa}{A \vdash \text{discard } e' \text{ in } e : \kappa}$$

and the analysis rule:

$$\text{Discard } \frac{A \vdash^{\wedge} e' : \kappa' \quad A \vdash^{\wedge} e : \kappa}{A \vdash^{\wedge} \text{discard}^{l} \ e' \text{ in } e : \kappa}$$

This gives an exact characterisation of the place where the analysis loses information. There are no reduction rules for 'discard'.

As with the 'either' construct, we let 'discard' expressions take over the label of the redex.

We want the non-standard reduction to be an even closer characterisation of the intersection flow analysis: if the flow analysis predicts a potential redex, we want this redex to be reduced by the semantics.

Pair-destruction "let (x, y) be e in e'" can *block* redexes. Consider

$$(\text{let } (x, y) \text{ be } z \text{ in } \lambda^{l_1} x.e)@e'$$

where our analysis will predict that the λ^{l_1} can be applied to e'. This will never be reduced by standard reduction. The 'discard', 'if' and 'either' constructs may block in a similar way. The solution to these problems is context propagation rules.

The reduction rules given in figure 6.7 and the context rules of figure 6.8 differ from standard reduction (as given in section 1.6) by the following properties:

1. As usual, an 'if' statement reduces if the conditional is True or False, but instead of rewriting to one branch it rewrites to *both*;

2. special cases for (β), (δ-let) and (δ-let-pair) make sure that no expression is discarded;

Contexts:

$$
\begin{array}{rcl}
C & ::= & [\,] \mid \lambda^l x.C \mid C@^l e \mid e@^l C \mid \\
 & & \mathrm{fix}^l x.C \mid \mathrm{let}^l\ x = C \text{ in } e \mid \mathrm{let}^l\ x = e \text{ in } C \mid \\
 & & \mathrm{if}^l\ C \text{ then } e' \text{ else } e'' \mid \mathrm{if}^l\ e \text{ then } C \text{ else } e'' \mid \mathrm{if}^l\ e \text{ then } e' \text{ else } C \mid \\
 & & (C, e')^l \mid (e, C)^l \mid \mathrm{let}^l\ (x, y) \text{ be } C \text{ in } e' \mid \mathrm{let}^l\ (x, y) \text{ be } e \text{ in } C \mid \\
 & & \mathrm{discard}^l\ C \text{ in } e' \mid \mathrm{discard}^l\ e \text{ in } C \mid \mathrm{either}^l\ C \text{ or } e \mid \mathrm{either}^l\ e \text{ or } C
\end{array}
$$

Reduction rules:

$$
\begin{array}{lrll}
(\beta) & (\lambda^{l'} x.e)@^l e' & \longrightarrow_s e[e'/x] & , \text{ if } x \in FV(e) \\
 & & \longrightarrow_s \mathrm{discard}^l\ e' \text{ in } e & , \text{ otherwise} \\
 & & & \\
(\delta\text{-if}) & \mathrm{if}^l\ \mathrm{True}^{l'} \text{ then } e \text{ else } e' & \longrightarrow_s \mathrm{either}^l\ e \text{ or } e' & \\
 & \mathrm{if}^l\ \mathrm{False}^{l'} \text{ then } e \text{ else } e' & \longrightarrow_s \mathrm{either}^l\ e' \text{ or } e & \\
 & & & \\
(\delta\text{-let}) & \mathrm{let}^l\ x = e \text{ in } e' & \longrightarrow_s e'[e/x] & , \text{ if } x \in FV(e') \\
 & & \longrightarrow_s \mathrm{discard}^l\ e \text{ in } e' & , \text{ otherwise} \\
 & & & \\
(\delta\text{-let-pair}) & & & \\
 & \mathrm{let}^l\ (x, y) \text{ be } (e, e') \text{ in } e'' & \longrightarrow_s e''[e/x][e'/y] & , \text{ if } x, y \in FV(e'') \\
 & & \longrightarrow_s \mathrm{discard}^l\ e' \text{ in } e''[e/x] & , \text{ if } x \in FV(e'') \text{ and } y \notin FV(e'') \\
 & & \longrightarrow_s \mathrm{discard}^l\ e \text{ in } e''[e'/y] & , \text{ if } y \in FV(e'') \text{ and } x \notin FV(e'') \\
 & & \longrightarrow_s \mathrm{discard}^l\ e \text{ in discard}^l\ e' \text{ in } e'' & , \text{ otherwise} \\
(\delta\text{-fix}) & \mathrm{fix}^l x.e & \longrightarrow_s e[\mathrm{fix}^l x.e/x] & \\
 & & & \\
 & C[e] & \longrightarrow_s C[e'] & , \text{ if } e \longrightarrow_s e'
\end{array}
$$

Figure 6.7: Non-standard reduction

Context rules:

(discard) $(\text{discard}^l\ e_1\ \text{in}\ e_2)@^{l'}e_3$
$\longrightarrow_s\ \text{discard}^l\ e_1\ \text{in}\ (e_2@^{l'}e_3)$
$\text{let}^{l'}\ (x_1, y_1)\ \text{be}\ (\text{discard}^l\ e_1\ \text{in}\ e_2)\ \text{in}\ e_3$
$\longrightarrow_s\ \text{discard}^l\ e_1\ \text{in}\ (\text{let}^{l'}\ (x_1, y_1)\ \text{be}\ e_2\ \text{in}\ e_3)$
$\text{if}^{l'}\ (\text{discard}^l\ e_1\ \text{in}\ e_2)\ \text{then}\ e_3\ \text{else}\ e_4$
$\longrightarrow_s\ \text{discard}^l\ e_1\ \text{in}\ (\text{if}^{l'}\ e_2\ \text{then}\ e_3\ \text{else}\ e_4)$

(pair) $(\text{let}^l\ (x, y)\ \text{be}\ e_1\ \text{in}\ e_2)@^{l'}e_3$
$\longrightarrow_s\ \text{let}^l\ (x, y)\ \text{be}\ e_1\ \text{in}\ (e_2@^{l'}e_3)$
$\text{let}^l\ (x_1, y_1)\ \text{be}\ (\text{let}^{l'}\ (x_2, y_2)\ \text{be}\ e_1\ \text{in}\ e_2)\ \text{in}\ e_3$
$\longrightarrow_s\ \text{let}^{l'}\ (x_2, y_2)\ \text{be}\ e_1\ \text{in}\ (\text{let}^l\ (x_1, y_1)\ \text{be}\ e_2\ \text{in}\ e_3)$
$\text{if}^{l'}\ (\text{let}^l\ (x, y)\ \text{be}\ e_1\ \text{in}\ e_2)\ \text{then}\ e_3\ \text{else}\ e_4$
$\longrightarrow_s\ \text{let}^l\ (x, y)\ \text{be}\ e_1\ \text{in}\ (\text{if}^{l'}\ e_2\ \text{then}\ e_3\ \text{else}\ e_4)$

(if) $(\text{if}^l\ e_1\ \text{then}\ e_2\ \text{else}\ e_3)@^{l'}e_4$
$\longrightarrow_s\ \text{if}^l\ e_1\ \text{then}\ (e_2@^{l'}e_4)\ \text{else}\ (e_3@^{l'}e_4)$
$\text{let}^{l'}\ (x, y)\ \text{be}\ (\text{if}^l\ e_2\ \text{then}\ e_3\ \text{else}\ e_4)\ \text{in}\ e_1$
$\longrightarrow_s\ \text{if}^l\ e_2$
$\text{then}\ (\text{let}^{l'}\ (x, y)\ \text{be}\ e_3\ \text{in}\ e_1)$
$\text{else}\ (\text{let}^{l'}\ (x, y)\ \text{be}\ e_4\ \text{in}\ e_1)$
$\text{if}^l\ (\text{if}^{l'}\ e_1\ \text{then}\ e_2\ \text{else}\ e_3)\ \text{then}\ e_4\ \text{else}\ e_5$
$\longrightarrow_s\ \text{if}^{l'}\ e_1$
$\text{then}\ (\text{if}^l\ e_2\ \text{then}\ e_4\ \text{else}\ e_5)$
$\text{else}\ (\text{if}^l\ e_3\ \text{then}\ e_4\ \text{else}\ e_5)$

(either) $(\text{either}^l\ e_1\ \text{or}\ e_2)@^{l'}e_3$
$\longrightarrow_s\ \text{either}^l\ (e_1@^{l'}e_3)\ \text{or}\ (e_2@^{l'}e_3)$
$\text{let}^{l'}\ (x, y)\ \text{be}\ (\text{either}^l\ e_2\ \text{or}\ e_3)\ \text{in}\ e_1$
$\longrightarrow_s\ \text{either}^l\ (\text{let}^{l'}\ (x, y)\ \text{be}\ e_2\ \text{in}\ e_1)\ \text{or}\ (\text{let}^{l'}\ (x, y)\ \text{be}\ e_3\ \text{in}\ e_1)$
$\text{if}^{l'}\ (\text{either}^l\ e_1\ \text{or}\ e_2)\ \text{then}\ e_3\ \text{else}\ e_4$
$\longrightarrow_s\ \text{either}^l\ (\text{if}^{l'}\ e_1\ \text{then}\ e_3\ \text{else}\ e_4)\ \text{or}\ (\text{if}^{l'}\ e_2\ \text{then}\ e_3\ \text{else}\ e_4)$

Figure 6.8: Non-standard reduction — context rules

3. there are context propagation rules for 'discard e in e'', 'if e then e' else e''', 'either e or e'' and 'let (x, y) be e in e''.

This gives an exact characterisation of the place where the analysis loses information. We include labels in the reduction rules.

Note that we used a similar characterisation of the strength of the polymorphic system: these were invariant under 'let' and 'fix' reduction. The characterisation was not complete, however, as we did not show exactly how and when information was lost.

Define the erasure $|e|$ of a term e to be the term where all occurrences of "discard e in e'" are replaced by e' and all occurrences of either e or e' are replaced by e. If $e \longrightarrow^* e'$ by standard reduction, then there exists e'' such that $e \longrightarrow_s^* e''$ and $|e''| = e'$. Thus, when we prove strong subject reduction for our non-standard reduction system, strong subject reduction for standard reduction follows.

6.4.1 Values

We will give a characterisation of normal forms under non-standard reduction:

Proposition 6.8 *Any expression v such that no e exists with $v \longrightarrow_s e$ is called a* value. *An expression v is a value if and only if it has the following syntax:*

$$\begin{array}{rcl} v & := & \text{let } (x,y) \textit{ be } \overline{v} \text{ in } v' \mid \text{if } \overline{v} \text{ then } v \text{ else } v' \mid \\ & & \text{either } v \text{ or } v' \mid \text{discard } v \text{ in } v' \mid \\ & & \lambda x.v \mid (v, v') \mid \text{True} \mid \text{False} \mid \overline{v} \\ \overline{v} & ::= & \overline{v}@v' \mid x \end{array}$$

Proof It is easy to see that if v has the above syntax, then no e exists such that $v \longrightarrow_s e$.

Let e be any expression. We will prove that if no e_0 exists such that $e \longrightarrow_s e_0$ then e has the above syntax. We prove this by induction on the structure of e. First we realise that e cannot be let $x = e'$ in e'' or fix $x.e$.

True: Is clearly a value.

False: Is clearly a value.

if e then e' else e'': Clearly, e, e' and e'' have to be values for 'if e then e' else e''' to be a value. Furthermore, e cannot be 'True' or 'False' since the conditional would then reduce to 'either e' or e''', and it cannot be a discard, a 'let $(x, y) \ldots$', another conditional or an either expression since then a context propagation rule would be applicable. Finally, it cannot be an abstraction or a pair since it would not be well-typed. The only things left are applications or variables.

either e or e': Clearly both e and e' have to be values.

x: Is clearly a value.

$\lambda x.e'$: Is a value if e' is.

$e'@e''$: Clearly e'' has to be a value. Also e' has to be a value. If e' is a pair or a truth value the expression is not well-typed. If e' is an abstraction, e cannot be a value. Similarly, if e' is a discard, a let $(x, y)\ldots$, a conditional or an either expression, one of the context propagation rules is applicable. The only options left for e' are a variable or an application.

(e', e''): Is a value if the components are.

let (x, y) be e' in e'': Both e' and e'' have to be values. If e' is a pair, e would not be a value and similarly if it is another pair destructor or a discard, a context rule would be applicable. Since e must be well typed, the only options left are applications and variables.

□

It is not hard to show the following properties for any e:

1. For any two non-standard reduction sequences reducing e to a value v, the same set of redexes are reduced.

2. If $e \longrightarrow_s^* e'$ lets destructor l consume constructor l' then there exists a reduction sequence using the standard rules plus (pair) and (if) context propagation rules where l consumes l'.

6.5 Subject Reduction

Proving subject reduction for the intersection type system is not much different from proving it for the previous systems.

To prove subject reduction under $\longrightarrow_s$ we need a substitution lemma as usual:

Lemma 6.9 (Substitution lemma) *If* $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{A, x : \kappa_2 \vdash^\wedge e_1 : \kappa_1}$ *and* $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{A \vdash^\wedge e_2 : \kappa_2}$ *then there exists* $\mathcal{T}_3$ *such that*

1. $\mathcal{T}_3 = \dfrac{\mathcal{T}_3'}{A \vdash^\wedge e_1[e_2/x] : \kappa_1}$ *and*

2. *For all* $l \in \text{Destructors}(e_1[e_2/x])$*:*

$$\mathcal{F}_{\mathcal{T}_3}(l) = \mathcal{F}_{\mathcal{T}_1}(l) \cup \mathcal{F}_{\mathcal{T}_2}(l)$$

Proof The lemma follows by simple induction on the structure of the derivation of $A, x : \kappa' \vdash^{\wedge} e : \kappa$. We give the ($\wedge$-I) case for illustration:

($\wedge$-I) Assume

$$\mathcal{T}_1 = \frac{\dfrac{\mathcal{T}_1'}{A, x : \kappa_2 \vdash^{\wedge} e_1 : \kappa_1} \qquad \dfrac{\mathcal{T}_1''}{A, x : \kappa_2 \vdash^{\wedge} e_1 : \kappa_1'}}{A, x : \kappa_2 \vdash^{\wedge} e : \kappa_1 \wedge \kappa_1'}$$

and

$$\mathcal{T}_2 = \frac{\mathcal{T}_2'}{A \vdash^{\wedge} e_2 : \kappa_2}$$

By induction there exists $\mathcal{T}_4$ and $\mathcal{T}_4$ such that

1. $\mathcal{T}_4 = \dfrac{\mathcal{T}_4'}{A \vdash^{\wedge} e_1[e_2/x] : \kappa_1}$
2. $\mathcal{T}_5 = \dfrac{\mathcal{T}_5'}{A \vdash^{\wedge} e_1[e_2/x] : \kappa_1'}$
3. For all $l \in \text{Destructors}(e_1[e_2/x])$:

$$\mathcal{F}_{\mathcal{T}_4}(l) = \mathcal{F}_{\frac{\mathcal{T}_1'}{A, x:\kappa_2 \vdash^{\wedge} e_1 : \kappa_1}}(l) \cup \mathcal{F}_{\mathcal{T}_2}(l)$$

4. For all $l \in \text{Destructors}(e_1[e_2/x])$:

$$\mathcal{F}_{\mathcal{T}_5}(l) = \mathcal{F}_{\frac{\mathcal{T}_1''}{A, x:\kappa_2 \vdash^{\wedge} e_1 : \kappa_1'}}(l) \cup \mathcal{F}_{\mathcal{T}_2}(l)$$

but then clearly we can construct

$$\mathcal{T}_3 = \frac{\dfrac{\mathcal{T}_4'}{A \vdash^{\wedge} e_1[e_2/x] : \kappa_1} \qquad \dfrac{\mathcal{T}_5'}{A \vdash^{\wedge} e_1[e_2/x] : \kappa_1'}}{A \vdash^{\wedge} e_1[e_2/x] : \kappa_1 \wedge \kappa_1'}$$

and obviously $l \in \text{Destructors}(e_1[e_2/x])$:

$$\begin{aligned}
\mathcal{F}_{\mathcal{T}_3}(l) &= \mathcal{F}_{\mathcal{T}_4}(l) \cup \mathcal{F}_{\mathcal{T}_5}(l) \\
&= \mathcal{F}_{\frac{\mathcal{T}_1'}{A, x:\kappa_2 \vdash^{\wedge} e_1 : \kappa_1}}(l) \cup \mathcal{F}_{\mathcal{T}_2}(l) \cup \mathcal{F}_{\frac{\mathcal{T}_1''}{A, x:\kappa_2 \vdash^{\wedge} e_1 : \kappa_1'}}(l) \cup \mathcal{F}_{\mathcal{T}_2}(l) \\
&= (\mathcal{F}_{\frac{\mathcal{T}_1'}{A, x:\kappa_2 \vdash^{\wedge} e_1 : \kappa_1}}(l) \cup \mathcal{F}_{\frac{\mathcal{T}_1''}{A, x:\kappa_2 \vdash^{\wedge} e_1 : \kappa_1'}}(l)) \cup \mathcal{F}_{\mathcal{T}_2}(l) \\
&= \mathcal{F}_{\mathcal{T}_1}(l) \cup \mathcal{F}_{\mathcal{T}_2}(l)
\end{aligned}$$

□

Theorem 6.10 (Subject Reduction) *If* $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{A \vdash^\wedge e_1 : \kappa}$ *and* $e_1 \longrightarrow e_2$ *then there exists* $\mathcal{T}_2$ *such that*

1. $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{A \vdash^\wedge e_2 : \kappa}$ *and*
2. *For all* $l \in \text{Destructors}(e_2)$ *we have*

$$\mathcal{F}_{\mathcal{T}_1}(l) = \mathcal{F}_{\mathcal{T}_2}(l)$$

and if $l \in \text{Destructors}(e_1)$ *consumes* $l' \in \text{Constructors}(e_1)$ *then*

$$\mathcal{F}_{\mathcal{T}_1}(l) = \mathcal{F}_{\mathcal{T}_2}(l) \cup \{l'\}$$

Proof The interesting cases follow from lemma 6.9. □

Soundness of flow functions computed from inference trees follows as usual:

Corollary 6.11 *Let* $\mathcal{T}$ *be any derivation for* e *and let* $A \vdash^\wedge e : \kappa$ *be its conclusion. Then* $\mathcal{F}_\mathcal{T} \models e$.

6.6 Subject Expansion

We will prove the subject expansion property. This can only hold if expansion preserves standard types, so this will be our implicit assumption throughout the section.

The subject expansion property we prove is *strong*, i.e. the flow computed is preserved by expansion.

We will often use the property that if

$$\frac{\mathcal{T}}{A, x : \kappa' \vdash^\wedge e : \kappa}$$

then

$$\frac{\mathcal{T}^*}{A, x : \kappa' \wedge \kappa'' \vdash^\wedge e : \kappa}$$

where $\mathcal{T}^*$ only differs from $\mathcal{T}$ in that bindings $x : \kappa'$ are replaced by $x : \kappa' \wedge \kappa''$ and use of the subsumption rule at occurrences of x.

The following lemma is an adaptation of lemma 5.23 to intersection types:

Lemma 6.12 *Let* e *be an expression with* $n > 0$ *occurrences of a variable* x. *If*

$$\mathcal{T}_{e[e'/x]} = \frac{\mathcal{T}'_{e[e'/x]}}{A \vdash^\wedge e[e'/x] : \kappa}$$

then there exists $\kappa_1 \cdots \kappa_n, \mathcal{T}'$ *and* $\mathcal{T}_1 \cdots \mathcal{T}_n$ *s.t.*

1. $\mathcal{T}_i = \dfrac{\mathcal{T}_i'}{A \vdash^\wedge e' : \kappa_i}$
2. $\mathcal{T}_e = \dfrac{\mathcal{T}_e'}{A, x : \bigwedge_i \kappa_i \vdash^\wedge e : \kappa}$ *and*
3. *For all* $l \in \text{Destructors}(e[e'/x])$*:*

$$\mathcal{F}_{\mathcal{T}_{e[e'/x]}}(l) = \mathcal{F}_{\mathcal{T}_e}(l) \cup \bigcup_i \mathcal{F}_{\mathcal{T}_i}(l)$$

Proof The proof is by induction on the derivation $\mathcal{T}_{e[e'/x]}$. Most cases are trivial — the only complication is that we have to treat subexpressions not containing x differently (we do not have to use induction on these). Furthermore, in the syntax directed rules, we first check if the expression of the conclusion is $x[e'/x]$ in which case the result follows immediately.

We will do the ($\wedge$-I) case as this is somewhat entertaining:

($\wedge$-I) Consider:

$$\mathcal{T}_{e[e'/x]} = \dfrac{\dfrac{\mathcal{T}'_{e[e'/x]}}{A \vdash^\wedge e[e'/x] : \kappa} \qquad \dfrac{\mathcal{T}''_{e[e'/x]}}{A \vdash^\wedge e[e'/x] : \kappa'}}{A \vdash^\wedge e[e'/x] : \kappa \wedge \kappa'}$$

By induction we have

$$\mathcal{T}_i' = \dfrac{\mathcal{T}_i^1}{A \vdash^\wedge e' : \kappa_i} \qquad \mathcal{T}_e' = \dfrac{\mathcal{T}_e^1}{A, x : \bigwedge_i \kappa_i \vdash^\wedge e : \kappa}$$

and

$$\mathcal{T}_i'' = \dfrac{\mathcal{T}_i^2}{A \vdash^\wedge e' : \kappa_i'} \qquad \mathcal{T}_e'' = \dfrac{\mathcal{T}_e^2}{A, x : \bigwedge_i \kappa_i' \vdash^\wedge e : \kappa'}$$

where for all $l \in \text{Destructors}(e[e'/x])$:

$$\mathcal{F}_{\frac{\mathcal{T}'_{e[e'/x]}}{A \vdash^\wedge e[e'/x] : \kappa}}(l) = \mathcal{F}_{\mathcal{T}_e'}(l) \cup \bigcup_i \mathcal{F}_{\mathcal{T}_i'}(l)$$

and

$$\mathcal{F}_{\frac{\mathcal{T}''_{e[e'/x]}}{A \vdash^\wedge e[e'/x] : \kappa'}}(l) = \mathcal{F}_{\mathcal{T}_e''}(l) \cup \bigcup_i \mathcal{F}_{\mathcal{T}_i''}(l)$$

By the ($\wedge$-I) rule we find

$$\mathcal{T}_i = \dfrac{\dfrac{\mathcal{T}_i^1}{A \vdash^\wedge e' : \kappa_i} \qquad \dfrac{\mathcal{T}_i^2}{A \vdash^\wedge e' : \kappa_i'}}{A \vdash^\wedge e' : \kappa_i \wedge \kappa_i'} \qquad (\dagger)$$

By the property of strengthened assumptions

$$\frac{\mathcal{T}_e^3}{A, x : \bigwedge_i \kappa_i \wedge \bigwedge_i \kappa_i' \vdash^\wedge e : \kappa}$$

and

$$\frac{\mathcal{T}_e^4}{A, x : \bigwedge_i \kappa_i \wedge \bigwedge_i \kappa_i' \vdash^\wedge e : \kappa'}$$

where $\mathcal{T}_e^3$ only differs from $\mathcal{T}_e^1$ in the binding for x and similarly for $\mathcal{T}_e^4$. We find

$$\mathcal{T}_e \dfrac{\dfrac{\mathcal{T}_e^3}{A, x : \bigwedge_i (\kappa_i \wedge \kappa_i') \vdash^\wedge e : \kappa} \quad \dfrac{\mathcal{T}_e^4}{A, x : \bigwedge_i (\kappa_i \wedge \kappa_i') \vdash^\wedge e : \kappa'}}{A, x : \bigwedge_i (\kappa_i \wedge \kappa_i') \vdash^\wedge e : \kappa \wedge \kappa'}$$

which with (†) constitutes *1.* and *2.* of the lemma.

$$\begin{aligned} \mathcal{F}_{\mathcal{T}_{e[e'/x]}}(l) &= \mathcal{F}_{\frac{\mathcal{T}'_{e[e'/x]}}{A \vdash^\wedge e[e'/x] : \kappa}}(l) \cup \mathcal{F}_{\frac{\mathcal{T}''_{e[e'/x]}}{A \vdash^\wedge e[e'/x] : \kappa'}}(l) \\ &= \mathcal{F}_{\mathcal{T}'_e}(l) \cup \bigcup_i \mathcal{F}_{\mathcal{T}'_i}(l) \cup \mathcal{F}_{\mathcal{T}''_e}(l) \cup \bigcup_i \mathcal{F}_{\mathcal{T}''_i}(l) \\ &= \mathcal{F}_{\mathcal{T}_e}(l) \cup \bigcup_i \mathcal{F}_{\mathcal{T}_i}(l) \end{aligned}$$

□

For standard intersection types it holds that a term is typable if (and only if) the term is normalising. The proof of this proceeds by proving that all normal forms are typable and that typability is preserved under beta-expansion. We will now prove subject expansion for $\longrightarrow_s$ along the same lines as the second part of this proof. The following theorem will not deal with reduction of 'fix' as the next section is devoted to this problem

Theorem 6.13 (Subject Expansion) *If* $\mathcal{T}_2 = \dfrac{\mathcal{T}_2'}{A \vdash^\wedge e_1 : \kappa}$ *and* $e_1 \longrightarrow_s e_2$ *then there exists* $\mathcal{T}_1$ *such that*

1. $\mathcal{T}_1 = \dfrac{\mathcal{T}_1'}{A \vdash^\wedge e_1 : \kappa}$
2. *For all* $l \in \text{Destructors}(e_2)$ *we have*

$$\mathcal{F}_{\mathcal{T}_1}(l) = \mathcal{F}_{\mathcal{T}_2}(l)$$

and if $l \in \text{Destructors}(e_1)$ *consumes* $l' \in \text{Constructors}(e_1)$ *then*

$$\mathcal{F}_{\mathcal{T}_1}(l) = \mathcal{F}_{\mathcal{T}_2}(l) \cup \{l'\}$$

Proof Induction over the definition of $\longrightarrow_s$, i.e. the β and δ rules are the base cases and the context rule is the induction step:

(β) Two cases:

1. Assume x not free in e. Then $(\lambda^{l'} x.e)@^l e' \longrightarrow_s \text{discard}^l\ e'$ in e. We have $\mathcal{T}_2 =$

$$\dfrac{\dfrac{\mathcal{T}_2'}{A \vdash^\wedge e' : \kappa'} \qquad \dfrac{\mathcal{T}_2''}{A \vdash^\wedge e : \kappa}}{A \vdash^\wedge \text{discard}^l\ e' \text{ in } e : \kappa}$$

   so we can clearly construct $\mathcal{T}_1 =$

$$\dfrac{\dfrac{\dfrac{\mathcal{T}_2'', x : \kappa'}{A, x : \kappa' \vdash^\wedge e : \kappa}}{A \vdash^\wedge \lambda^{l'} x.e : \kappa' \to^{\{l'\}} \kappa} \qquad \dfrac{\mathcal{T}_2'}{A \vdash^\wedge e' : \kappa'}}{A \vdash^\wedge (\lambda^{l'} x.e)@^l e' : \kappa}$$

   Point *2.* follows immediately.

2. Assume x has $n \geq 1$ occurrences in e. Then $(\lambda^{l'} x.e)@^l e' \longrightarrow_s e[e'/x]$. By assumption $\mathcal{T}_2 =$

$$\dfrac{\mathcal{T}_2'}{A \vdash^\wedge e[e'/x] : \kappa}$$

   By lemma 6.12 we have $\mathcal{T}_i'', \kappa_i$ and $\mathcal{T}$ s.t.

$$\mathcal{T}_i'' = \dfrac{\mathcal{T}_i'''}{A \vdash^\wedge e' : \kappa_i} \qquad \text{and} \qquad \mathcal{T} = \dfrac{\mathcal{T}'}{A, x : \bigwedge_i \kappa_i \vdash^\wedge e : \kappa}$$

   We construct $\mathcal{T}_1 =$

$$\dfrac{\dfrac{\dfrac{\mathcal{T}'}{A, x : \bigwedge_i \kappa_i \vdash^\wedge e : \kappa}}{A \vdash^\wedge \lambda^{l'} x.e : (\bigwedge_i \kappa_i) \to^{\{l'\}} \kappa} \qquad \dfrac{\dfrac{\mathcal{T}_1'''}{A \vdash^\wedge e' : \kappa_1} \quad \cdots \quad \dfrac{\mathcal{T}_n'''}{A \vdash^\wedge e' : \kappa_n}}{A \vdash^\wedge e' : \bigwedge_i \kappa_i}}{A \vdash^\wedge (\lambda^{l'} x.e)@^l e' : \kappa}$$

   Point *2.* follows from lemma 6.12.

(δ-if) Assume

$$\text{if}^l\ \text{True}^{l'}\ \text{then } e \text{ else } e' \longrightarrow_s \text{either}^l\ e \text{ or } e'$$

By assumption $\mathcal{T}_2 =$

$$\dfrac{\dfrac{\mathcal{T}_2'}{A \vdash^\wedge e : \kappa} \qquad \dfrac{\mathcal{T}_2''}{A \vdash^\wedge e' : \kappa}}{A \vdash^\wedge \text{either}^l\ e \text{ or } e' : \kappa}$$

So we can construct $\mathcal{T}_1 =$

$$\dfrac{\dfrac{}{A \vdash^\wedge \text{True}^{l'} : \text{Bool}^{\{l'\}}} \qquad \dfrac{\mathcal{T}_2'}{A \vdash^\wedge e : \kappa} \qquad \dfrac{\mathcal{T}_2''}{A \vdash^\wedge e' : \kappa}}{A \vdash^\wedge \text{if}^l\ \text{True}^{l'} \text{ then } e \text{ else } e' : \kappa}$$

The case for False is similar.

(δ-let-pair) The four cases are handled as in the (β) case.

"Context Rule": Trivial induction on the context C.

"Context propagation rules": The (discard) and (pair) cases follow from a simple rearrangement of the derivations. The (if) and (either) cases requires the use of $\wedge$ but are relatively straightforward: we give the first case of (if) for illustration:

Assume

$$(\text{if}^l\ e_1 \text{ then } e_2 \text{ else } e_3)@^{l'} e_4 \longrightarrow_s \text{if}^l\ e_1 \text{ then } (e_2@^{l'} e_4) \text{ else } (e_3@^{l'} e_4)$$

Without loss of generality, we can assume $\mathcal{T}_2 =$

$$\dfrac{\dfrac{\mathcal{T}_2'}{A \vdash^\wedge e_1 : \text{Bool}^{L_1}} \qquad \mathcal{T}_3 \qquad \mathcal{T}_4}{A \vdash^\wedge \text{if}^l\ e_1 \text{ then } (e_2@^{l'} e_4) \text{ else } (e_3@^{l'} e_4) : \kappa}$$

where $\mathcal{T}_3 =$

$$\dfrac{\dfrac{\dfrac{\mathcal{T}_3'}{A \vdash^\wedge e_2 : \kappa_4 \to^{L_2} \kappa_2} \qquad \dfrac{\mathcal{T}_3''}{A \vdash^\wedge e_4 : \kappa_4}}{A \vdash^\wedge (e_2@^{l'} e_4) : \kappa_2} \qquad \vdash^\wedge \kappa_2 \leq \kappa}{A \vdash^\wedge (e_2@^{l'} e_4) : \kappa}$$

and $\mathcal{T}_4 =$

$$\dfrac{\dfrac{\dfrac{\mathcal{T}_4'}{A \vdash^\wedge e_3 : \kappa_4' \to^{L_3} \kappa_3} \qquad \dfrac{\mathcal{T}_4''}{A \vdash^\wedge e_4 : \kappa_4'}}{A \vdash^\wedge (e_3@^{l'} e_4) : \kappa_3} \qquad \vdash^\wedge \kappa_3 \leq \kappa}{A \vdash^\wedge (e_3@^{l'} e_4) : \kappa}$$

We can now construct $\mathcal{T}_1 =$

$$\dfrac{\dfrac{\dfrac{\mathcal{T}_2'}{A \vdash^\wedge e_1 : \mathrm{Bool}^{L_1}} \quad \mathcal{T}_5 \quad \mathcal{T}_6}{A \vdash^\wedge \mathrm{if}^l\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3 : (\kappa_4 \wedge \kappa_4') \to^{L_2 \cup L_3} \kappa} \quad \dfrac{\dfrac{\mathcal{T}_3''}{A \vdash^\wedge e_4 : \kappa_4} \quad \dfrac{\mathcal{T}_4''}{A \vdash^\wedge e_4 : \kappa_4'}}{A \vdash^\wedge e_4 : \kappa_4 \wedge \kappa_4'}}{A \vdash^\wedge (\mathrm{if}^l\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3)@^{l'} e_4 : \kappa}$$

where $\mathcal{T}_5 =$

$$\dfrac{\dfrac{\mathcal{T}_3'}{A \vdash^\wedge e_2 : \kappa_4 \to^{L_2} \kappa_2} \quad \dfrac{\overline{\vdash^\wedge \kappa_4 \wedge \kappa_4' \le \kappa_4} \quad \vdash^\wedge \kappa_2 \le \kappa \quad L_2 \subseteq L_2 \cup L_3}{\vdash^\wedge \kappa_4 \to^{L_2} \kappa_2 \le (\kappa_4 \wedge \kappa_4') \to^{L_2 \cup L_3} \kappa}}{A \vdash^\wedge e_2 : (\kappa_4 \wedge \kappa_4') \to^{L_2 \cup L_3} \kappa}$$

and $\mathcal{T}_6 =$

$$\dfrac{\dfrac{\mathcal{T}_4'}{A \vdash^\wedge e_3 : \kappa_4' \to^{L_3} \kappa_3} \quad \dfrac{\overline{\vdash^\wedge \kappa_4 \wedge \kappa_4' \le \kappa_4'} \quad \vdash^\wedge \kappa_3 \le \kappa \quad L_3 \subseteq L_2 \cup L_3}{\vdash^\wedge \kappa_4' \to^{L_3} \kappa_3 \le (\kappa_4 \wedge \kappa_4') \to^{L_2 \cup L_3} \kappa}}{A \vdash^\wedge e_3 : (\kappa_4 \wedge \kappa_4') \to^{L_2 \cup L_3} \kappa}$$

□

6.7 Handling 'fix'

The idea of this section is to show that there exists a finite unfolding e^m of any expression $\mathrm{fix}\, x.e$ such that the analysis is invariant under expansion of the reduction rule $\mathrm{fix}\, x.e \longrightarrow_s e^m$ (throughout this section, we will assume that x occurs in e, since the problem is trivial otherwise). The strategy is as follows:

1. Show that it is no restriction to consider expressions $\mathrm{fix}\, x.e$ with exactly *one* occurrence of x (lemma 6.14).
2. Show subject expansion for $C[\mathrm{fix}\, x.e^{(m)}] \longrightarrow C[e^n]$ where $e^{(m)}$ is m times unfolding of e and e^n is n unfoldings of e and $\bot$ inserted for x (corollary 6.18).
3. Show subject expansion for $C[\mathrm{fix}\, x.e] \longrightarrow C[\mathrm{fix}\, x.e^{(m)}]$ (lemma 6.19).

The following lemma is a trivial consequence of theorem 6.13:

Lemma 6.14 *Let e be an expression with $n > 0$ occurrences of x and no occurrences of y. Let*

$$\dfrac{\mathcal{T}}{A \vdash^\wedge C[e] : \kappa} \quad \textit{and} \quad \dfrac{\mathcal{T}'}{A' \vdash^\wedge C[(\lambda^{l'} y.e[y/x])@^l x] : \kappa}$$

be minimal derivations. Then

$$\mathcal{F}_{\frac{\mathcal{T}}{A\vdash^{\wedge} C[e]:\kappa}}(l) \cup \{l'\} = \mathcal{F}_{\frac{\mathcal{T}'}{A'\vdash^{\wedge} C[(\lambda y.e[y/x])@x]:\kappa}}(l)$$

and for all $l'' \in \text{Destructors}(C[e])$, $l'' \neq l$ *implies*

$$\mathcal{F}_{\frac{\mathcal{T}}{A\vdash^{\wedge} C[e]:\kappa}}(l'') = \mathcal{F}_{\frac{\mathcal{T}'}{A'\vdash^{\wedge} C[(\lambda y.e[y/x])@x]:\kappa}}(l'')$$

Proof Trivial consequence of theorem 6.13. □

Definition 6.15 *For any expression* e *of type* t*, define*

$$\begin{aligned} e^0 &= \bot_t \\ e^{n+1} &= e[e^n/x] \end{aligned}$$

Lemma 6.16 *Let* e *and* C *be given such that* e *has exactly one occurrence of* x*. Then there exists* m *and* $n < m$ *such that if*

$$\frac{\mathcal{T}}{A \vdash^{\wedge} C[e^m] : \kappa}$$

is a minimal derivation then it contains the judgements $A' \vdash^{\wedge} e^n : \kappa'$ *and* $A'' \vdash^{\wedge} e^m : \kappa''$ *with* $A' \mid_{FV(e)} = A'' \mid_{FV(e)}$ *and* $\kappa' = \kappa''$*.*

Proof Clearly, for every x free in e, the underlying standard types of $A'(x)$ and $A''(x)$ are the same. Similarly, the underlying types of κ' and κ'' are the same. By finiteness of $\mathcal{K}(t)/=$ there are only finitely many pairs $(A \mid_{FV(e)}, \kappa)$ and hence there must exist some m where we meet a judgement, that has occurred earlier in the derivation.

□

Definition 6.17 *For any expression* e*, define*

$$\begin{aligned} e^{(0)} &= x \\ e^{(n+1)} &= e[e^{(n)}/x] \end{aligned}$$

Corollary 6.18 *Let* m, n *be as computed by lemma 6.16 and*

$$\frac{\mathcal{T}}{A \vdash^{\wedge} C[e^m] : \kappa}$$

be the minimal derivation. There exists

$$\frac{\mathcal{T}'}{A \vdash C[\text{fix}\, x.e^{(m-n)}] : \kappa}$$

such that for all $l \in \text{Destructors}(C[e^m])$*:*

$$\mathcal{F}_{\frac{\mathcal{T}}{A\vdash^{\wedge} C[\text{fix}\, x.e^{(m-n)}]:\kappa}}(l) \subseteq \mathcal{F}_{\frac{\mathcal{T}'}{A\vdash^{\wedge} C[e^m]:\kappa}}(l)$$

Proof Immediate from lemma 6.16 and lemma 5.23. □

Lemma 6.19 *Let $\mathcal{T}$, A, κ, C and e (with exactly one occurrence of x) be given such that*

$$\frac{\mathcal{T}}{A \vdash^{\wedge} C[\mathrm{fix}\, x.e^{(m)}] : \kappa}$$

is a minimal derivation. Then there exists $\mathcal{T}'$ such that

$$\frac{\mathcal{T}'}{A \vdash^{\wedge} C[\mathrm{fix}\, x.e] : \kappa}$$

and for all $l \in \mathrm{Destructors}(C[e^m])$:

$$\mathcal{F}_{\frac{\mathcal{T}'}{A \vdash^{\wedge} C[\mathrm{fix}\, x.e] : \kappa}}(l) \subseteq \mathcal{F}_{\frac{\mathcal{T}}{A \vdash^{\wedge} C[\mathrm{fix}\, x.e^{(m)}] : \kappa}}(l)$$

Proof We have the following derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{\mathcal{T}_0}{A, x : \kappa_m \vdash^{\wedge} e^{(0)} : \kappa_0}}{\cdots}}{A, x : \kappa_m \vdash^{\wedge} e^{(m)} : \kappa_m}}{\cfrac{A \vdash^{\wedge} \mathrm{fix}\, x.e^{(m)} : \kappa_m}{A \vdash^{\wedge} C[\mathrm{fix}\, x.e^{(m)}] : \kappa}}$$

Dismantle this derivation according to lemma 5.23 into

$$\frac{\mathcal{T}_0}{A, x : \kappa_m \vdash^{\wedge} e : \kappa_0} \qquad \frac{\mathcal{T}_1}{A, x : \kappa_0 \vdash^{\wedge} e : \kappa_1} \quad \cdots \quad \frac{\mathcal{T}_m}{A, x : \kappa_{m-1} \vdash^{\wedge} e : \kappa_m}$$

By the property of strengthened assumptions, we find

$$\frac{\mathcal{T}_0'}{A, x : \bigwedge_i \kappa_i \vdash^{\wedge} e : \kappa_0} \qquad \frac{\mathcal{T}_1'}{A, x : \bigwedge_i \kappa_i \vdash^{\wedge} e : \kappa_1} \quad \cdots \quad \frac{\mathcal{T}_m'}{A, x : \bigwedge_i \kappa_i \vdash^{\wedge} e : \kappa_m}$$

(where $\mathcal{T}_i'$ only differs from $\mathcal{T}_i'$ in the assumptions for x). Then by ($\wedge$-I) we have

$$\cfrac{\cfrac{\cfrac{\cfrac{\mathcal{T}_0'}{A, x : \bigwedge_i \kappa_i \vdash^{\wedge} e : \kappa_0} \quad \cdots \quad \cfrac{\mathcal{T}_m'}{A, x : \bigwedge_i \kappa_i \vdash^{\wedge} e : \kappa_m}}{A, x : \bigwedge_i \kappa_i \vdash^{\wedge} e : \bigwedge_i \kappa_i}}{A \vdash^{\wedge} \mathrm{fix}\, x.e : \bigwedge_i \kappa_i} \qquad \vdash^{\wedge} \bigwedge_i \kappa_i \leq \kappa_m}{\cfrac{A \vdash^{\wedge} \mathrm{fix}\, x.e : \kappa_m}{A \vdash^{\wedge} C[\mathrm{fix}\, x.e] : \kappa}}$$

Invariance of the computed flow relations follows by the construction.

□

We summarize corollary 6.18 and lemma 6.19 as follows (where we use lemma 6.14 to generalise to an arbitrary number of occurrences of the fix-bound variable).

Theorem 6.20 *Let $C[\text{fix}\, x.e]$ be given. Then there exists m such that if*

$$\frac{\mathcal{T}}{A \vdash^{\wedge} C[e^m] : \kappa}$$

then there exists $\mathcal{T}'$ such that

$$\frac{\mathcal{T}'}{A \vdash^{\wedge} C[\text{fix}\, x.e] : \kappa}$$

and for all $l \in \text{Destructors}(C[e^m])$:

$$\mathcal{F}_{\frac{\mathcal{T}'}{A \vdash^{\wedge} C[\text{fix}^l\, x.e] : \kappa}}(l) \subseteq \mathcal{F}_{\frac{\mathcal{T}}{A \vdash^{\wedge} C[e^m] : \kappa}}(l)$$

6.8 Flow in Normal Forms

We will now prove the main theorem for normal forms. We will use the notation $A \wedge A'$ for the environment mapping x to $A(x) \wedge A'(x)$ if x is in the domain of both A and A', and to $A(x)$ resp. $A'(x)$ if x is not in the domain of A' resp. not in the domain of A. We extend this to use the abbreviation $\mathcal{T} \wedge A$ for the derivation where A is intersected with all environments in $\mathcal{T}$ (this derivation contains appropriate subsumption steps at variable occurrences). Clearly $\mathcal{T} \wedge A$ is a valid derivation if $\mathcal{T}$ is.

Theorem 6.21 *If v is a value, there exists $A, \kappa, \mathcal{T}$ such that*

$$\mathcal{T} = \frac{\mathcal{T}'}{A \vdash^{\wedge} v : \kappa}$$

and $\mathcal{F}_{\mathcal{T}}(l) = \{\}$ for all $l \in \text{Destructors}(v)$.

Proof Call a property $\kappa \in \mathcal{K}$ *result-empty* iff all positively occurring labels are the empty set $\{\}$. Similarly, κ is called *argument-empty* iff all negatively occurring labels are the empty set $\{\}$.

We will show that that

1. For all v not being variables, applications or bottom there exists $A, \kappa, \mathcal{T}$ such that

   (a) $\dfrac{\mathcal{T}}{A \vdash^{\wedge} v : \kappa}$

(b) $A(x)$ result-empty for all x

(c) κ argument-empty.

2. For all $\overline{v}$ and result-empty κ there exists $A, \mathcal{T}$ such that

(a) $\dfrac{\mathcal{T}}{A \vdash^{\wedge} \overline{v} : \kappa}$

(b) $A(x)$ result-empty for all x

It follows from point 2(a) that any expression occurring in a "consumption" context can be given a type where all positively occurring annotations (in particular the top annotation) is the empty set.

The proof proceeds by induction over the structure of values.

True^l: Any A, κ will do.

False^l: Any A, κ will do.

x: Let κ be the given result empty type. Then any $(A, x : \kappa)$ will do.

$\lambda^l x.v$: By induction we have

$$\frac{\mathcal{T}}{A, x : \kappa \vdash^{\wedge} v : \kappa'}$$

where A and κ are result empty. If v is not a variable or an application, κ' is argument empty. Otherwise, the above is true for *any* result-empty κ' in particular the one that is also argument empty. Thus $\kappa \to^L \kappa'$ is argument-empty for any L. So

$$\frac{\dfrac{\mathcal{T}}{A, x : \kappa \vdash^{\wedge} v : \kappa'}}{A \vdash^{\wedge} \lambda^l x.v : \kappa \to^L \kappa'}$$

is the sought after derivation.

$\overline{v}@^l v'$: If v' is not a variable or an application, we have by induction

$$\frac{\mathcal{T}}{A' \vdash^{\wedge} v' : \kappa'}$$

where A' is result-empty and κ' argument-empty. If v' is a variable or an application, then the above is true for any result-empty κ' and in particular for the argument- and result-empty κ'.

Let any result-empty κ be given. Then $\kappa' \to^{\{\}} \kappa$ is also result-empty. Then by induction there is

$$\frac{\mathcal{T}}{A \vdash^{\wedge} \overline{v} : \kappa' \to^{\{\}} \kappa}$$

where A is result-empty.

We now construct

$$\frac{\mathcal{T} \wedge A' \qquad \mathcal{T}' \wedge A}{A \wedge A' \vdash^{\wedge} \overline{v} : \kappa' \to^{\{\}} \kappa \qquad A \wedge A' \vdash^{\wedge} v' : \kappa'}$$
$$A \wedge A' \vdash^{\wedge} \overline{v} @^{l} v' : \kappa$$

where $A \wedge A'$ is result-empty and κ is any given result-empty type.

if^{l} $\overline{v}$ then v' else v'': By induction we find $A, \mathcal{T}$ s.t.

$$\frac{\mathcal{T}}{A \vdash^{\wedge} \overline{v} : \mathrm{Bool}^{\{\}}}$$

and $A', A'', \kappa', \kappa'', \mathcal{T}', \mathcal{T}''$ s.t.

$$\frac{\mathcal{T}'}{A' \vdash^{\wedge} v' : \kappa'} \qquad \text{and} \qquad \frac{\mathcal{T}''}{A'' \vdash^{\wedge} v'' : \kappa''}$$

where A, A' and A'' are result-empty and κ', κ'' are argument-empty. Let $A''' = A \wedge A' \wedge A''$. Clearly, there exists an argument empty type κ''' such that

$$\frac{\mathcal{T} \wedge A' \wedge A'' \qquad \dfrac{\mathcal{T}' \wedge A \wedge A''}{A''' \vdash^{\wedge} v' : \kappa'} \quad \vdash^{\wedge} \kappa' \leq \kappa''' \qquad \dfrac{\mathcal{T}'' \wedge A \wedge A'}{A''' \vdash^{\wedge} v'' : \kappa''} \quad \vdash^{\wedge} \kappa'' \leq \kappa'''}{A''' \vdash^{\wedge} \overline{v} : \mathrm{Bool}^{\{\}} \qquad A''' \vdash^{\wedge} v' : \kappa''' \qquad A''' \vdash^{\wedge} v'' : \kappa'''}$$
$$A''' \vdash^{\wedge} \text{if } \overline{v} \text{ then } v' \text{ else } v'' : \kappa'''$$

either^{l} v or v': Similar, but simpler than the 'if' case.

$(v, v')^{l}$: Follows by simple induction.

let^{l} (x, y) be $\overline{v}$ in v': We have by induction

$$\frac{\mathcal{T}'}{A', x : \kappa_x, y : \kappa_y \vdash^{\wedge} v' : \kappa'}$$

where $A', x : \kappa_x, y : \kappa_y$ is result-empty and κ argument-empty (if v is an application or a variable we choose the argument- and result-empty κ as above).

For the result-empty $\kappa_x \times^{\{\}} \kappa_y$ we have

$$\frac{\mathcal{T}}{A \vdash^{\wedge} \overline{v} : \kappa_x \times^{\{\}} \kappa_y}$$

where A is result-empty.

We conclude

$$\frac{\dfrac{\mathcal{T} \wedge A'}{A \wedge A' \vdash^{\wedge} \overline{v} : \kappa_x \to^{\{\}} \kappa_y} \qquad \dfrac{\mathcal{T}' \wedge A}{(A \wedge A'), x : \kappa_x, y : \kappa_y \vdash^{\wedge} v' : \kappa'}}{A \wedge A' \vdash^{\wedge} \text{let}^{l}\ (x, y)\ \text{be}\ \overline{v}\ \text{in}\ v' : \kappa'}$$

where clearly $A \wedge A'$ is result-empty and κ result empty.

discard^{l} v in v': Simple induction.

$\bot_t$: Clearly, $A \vdash^{\wedge} \bot_t : \kappa$ for any A, κ.

□

6.9 Summarising the Results

Sections 6.6, 6.7 and 6.8 prove that the analysis is exact under non-standard reduction: let e be any expression, apply theorem 6.20 exhaustively yielding a fix-free term e'. By theorem 6.10 and theorem 6.20, the minimal predictable flow of e and e' is identical. By inductively applying theorem 6.13, we have that the minimal predictable flow for e' represent exactly the redexes met when reducing e' to a value.

If non-standard reduction reduces e to a value v such that l consumes l' then there *exists* a reduction sequence using standard reduction extended with context propagation rules for (pair) and (if) such that l consumes l'. (Note that the context propagation rules are necessary since e.g. (let (a, b) be x in $\lambda y.y$)@True is a value under standard reduction — for closed terms, context propagation rules are not necessary). We can summarize:

Theorem 6.22 (Exactness) *Let e be any expression and let $\mathcal{T}$ be the minimal derivation for e. Then for any redex $l \in$ Destructors(e) and any $l' \in \mathcal{F}_{\mathcal{T}}(l)$ there exists a reduction sequence using standard reduction plus the context propagation rules (pair) and (if) such that l consumes l'.*

Chapter 7

Shivers' CFA

The subtyping flow analysis of chapter 3 (which we showed corresponds to Sestoft's [Ses88]) is often referred to as 0CFA. The term 0CFA, however, originates from Olin Shivers' thesis [Shi91c] where a family of analyses nCFA for every n is defined. As we will see in this section, Shivers' 0CFA does *not* correspond to Sestoft's analysis, but is in fact strictly more powerful.

Section 7.1 presents Shivers' 0CFA in detail and compares it with closure analysis. Section 7.2 briefly describes the generalisation to nCFA. Finally, section 7.3 describes 0CFA as originally defined by Shivers for a tail-recursive language — the change of language gives rise to certain optimisations of the algorithm at the cost of some precision (though applying this analysis to the CPS-transform of an expression still potentially gives better results than closure analysis).

7.1 0CFA

Shivers defines his analysis for a CPS language (tail recursive); this makes direct comparison with Sestoft's analysis difficult (this might be one reason for the confusion). In figure 7.1 we try to present 0CFA for our language while being as faithful as possible to the original idea. It *does* make a difference that Shivers is defining his analysis for a tail-recursive language, we will return to this in section 7.3.

We have left out the pair construct as this does not fit smoothly into the formulation. We will return to this below.

An environment maps variables to the set of labels, that the variable can be bound to. We define the union of two environments by $(\rho \cup \rho')(x) = \rho(x) \cup \rho'(x)$. During analysis, the assumption on each variable x in the environment ρ is made less precise by making $\rho(x)$ bigger.

The analysis function C takes an expression and an environment as arguments and returns a pair. The first component of this pair is the set of labels to which the expression can evaluate. The second component is an

$$C : \mathrm{Exp} \to (\mathrm{Var} \to \mathcal{P}(\mathcal{L})) \to \mathcal{L} \times (\mathrm{Var} \to \mathcal{P}(\mathcal{L}))$$

$$\begin{array}{rcl}
C[\![x]\!]\rho & = & (\rho(x), \rho) \\
C[\![\lambda^l x.e]\!]\rho & = & (\{l\}, \rho) \\
C[\![\mathrm{True}^l]\!]\rho & = & (\{l\}, \rho) \\
C[\![\mathrm{False}^l]\!]\rho & = & (\{l\}, \rho) \\
C[\![\mathrm{if}^l\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3]\!]\rho & = & \mathrm{let}\ (L_1, \rho_1) = C[\![e_1]\!]\rho \\
 & & \mathrm{let}\ (L_2, \rho_2) = C[\![e_2]\!]\rho \\
 & & \mathrm{let}\ (L_3, \rho_3) = C[\![e_3]\!]\rho \\
 & & \mathrm{in}\ (L_2 \cup L_3, \rho_1 \cup \rho_2 \cup \rho_3) \\
C[\![\mathrm{fix}\, x.e]\!]\rho & = & \mathit{fix}(L, \rho').(C[\![e]\!](\rho \cup \rho' \cup [x \mapsto L])) \\
C[\![\mathrm{let}\ x = e_1\ \mathrm{in}\ e_2]\!]\rho & = & \mathrm{let}\ (L_1, \rho_1) = C[\![e_1]\!]\rho \\
 & & \mathrm{let}\ (L_2, \rho_2) = C[\![e_2]\!](\rho_1 \cup [x \mapsto L_1]) \\
 & & \mathrm{in}\ (L_2, \rho_2) \\
C[\![e_1 @^l e_2]\!]\rho & = & \mathrm{let}\ (\{l_1, \cdots, l_n\}, \rho_1) = C[\![e_1]\!]\rho \\
 & & \mathrm{let}\ (L, \rho_2) = C[\![e_2]\!]\rho \\
 & & \mathrm{let}\ \rho_3 = \rho_1 \cup \rho_2 \\
 & & \mathrm{let}\ (L_i, \rho_i) \\
 & & \qquad = C[\![\mathit{bodyof}\,(l_i)]\!](\rho_3 \cup [\mathit{varof}\,(l_i) \mapsto L]) \\
 & & \mathrm{in}\ (\bigcup_{i \in \{1,\cdots,n\}} L_i, \bigcup_{i \in \{1,\cdots,n\}} \rho_i)
\end{array}$$

Figure 7.1: 0CFA

updated environment. Functions *bodyof* and *varof* picks out the body resp. bound variable of an abstraction given its label. It should be clear that the analysis can compute a flow function as a side effect.

The variable case looks up the variable in the environment and returns this value together with the unaltered environment. Analysing a value returns the unaltered environment and the label of the value: in particular the body of an abstraction is not analysed and will only be so when the abstraction is applied. Conditionals are analysed by analysing the three subexpressions under the same assumptions. The result is the union of the labels of the branches and the union of all updated environments. That the subexpressions are analysed under the same assumptions reflect that updates performed when evaluating one subexpression cannot affect the others. Fix-expressions are analysed by computing the least fixed point for the result of analysing the expression. If we assume that the body of a 'fix'-bound is always a lambda the following simpler definition suffices:

$$\begin{array}{rcl} C[\![\mathrm{fix}\, x.e]\!]\rho & = & \mathrm{let}\ (L,\rho') = C[\![e]\!]\rho \\ & & \mathrm{in}\ (L, \rho'[x \mapsto L]) \end{array}$$

In the let-case, we first analyse the let-bound expression e_1. This results in a set of labels L_1 and a new environment ρ_1. We then analyse the body e_2 of the let in the environment $\rho_1 \cup [x \mapsto L_1]$. It is obvious that the binding for x needs to be updated, but less obvious that we need the environment ρ_1 (and not just ρ): the reason is that variables that were bound when analysing e_1 might still be alive and could be referenced while analysing e_2. This is made clear by the following example:

Example 7.1 Consider

$$\begin{array}{l} \mathrm{let}\ f = (\lambda^{l_1} x.\lambda^{l_2} y.x)@\mathrm{True}^{l_3} \\ \mathrm{in}\ f@\mathrm{False} \end{array}$$

Analysing the let-bound expressions will return $(\{l_2\}, [x \mapsto \{l_3\}])$. When we analyse the body, we find that f can be l_2 and the analysis proceeds to analyse the body of λ^{l_2}. It is then important that the binding of x to $\{l_3\}$ is alive.

□

We now come to the most important and difficult case: application. When analysing an application $e@^l e'$, we first compute the set of functions $\{l_1, \cdots, l_n\}$ that e can evaluate to and the set of values L that the argument can evaluate to. Analysis of both expressions is done in the given environment ρ. For each of the functions $\lambda^{l_i} x_i.e_i$ we analyse e_i under assumption that x_i is bound to L. This is done in an environment $\rho_1 \cup \rho_2$ since analysing e and e' might have bound variables that are live when analysing the e_i. This is illustrated by the following example:

Example 7.2 Consider:

$$((\lambda^{l_1} x.\lambda^{l_2} f.f@x)@\text{True})@^{l_5}((\lambda^{l_3} y.\lambda^{l_4} z.y)@\text{False})$$

When analysing $@^{l_5}$ we find that the functions applied are $\{l_2\}$ and the arguments are $\{l_4\}$. Analysing $f@x$ under assumption that f is $\{l_4\}$ will clearly both reference x and y.

□

The result of analysing an application is the union of the labels computed by analysing the function bodies and the union of the environments computed.

The main difference to Sestoft's analysis is in the application case where the analysis proceeds to analyse the bodies of all functions that potentially are applicable. Furthermore, Sestoft computes a fixed-point for the environment: all lookups are intuitively done in the same environment. In 0CFA lookups are performed in a decreasingly precise environment. It is not hard to see that 0CFA is at least as strong as Sestoft's closure analysis.

A simple example shows how 0CFA can yield better results than sub-type based flow analysis:

Example 7.3 Consider

$$\begin{array}{l}\text{let } id = \lambda x.x \\ \text{in } \text{if } id@\text{True}^{l_1} \\ \quad \text{then } id@\text{False}^{l_2} \\ \quad \text{else True}^{l_3}\end{array}$$

In the first application of id the environment is updated to map x to $\{l_1\}$. Since the body of the lambda is analysed at this point, we find that the conditional can only evaluate to l_1. In the second application, the environment is updated to map x to $\{l_2\}$ so the result is also precise for this application. The environment returned will map x to $\{l_1, l_2\}$. The result of the whole expression will be $\{l_2, l_3\}$.

□

Of course the analysis is not exact as shown by the following example:

Example 7.4 Consider

$$\begin{array}{l}\text{let } id = \lambda x.x \\ \text{in } (\lambda^{l_3} y.id@\text{False}^{l_2})@^{l_4}(id@\text{True}^{l_1})\end{array}$$

When we analyse application l_4 we find that the function can be $\{l_3\}$. Analysing the argument updates the binding for x to $\{l_1\}$ and returns that the argument to $@^{l_4}$ can be $\{l_1\}$. We proceed to analyse the body of λ^{l_3} in an environment where *both* x and y are bound to $\{l_1\}$. Thus this application of id will update the binding for x to $\{l_1, l_2\}$ which will also be the result of the application and thus of the whole expression.

□

This example suggests that we could update environments *destructively* instead of taking the union with the already present binding. This would, however, not be safe as the next example shows:

Example 7.5 Consider the following expression:

$$\begin{array}{l}\text{let } app = \lambda f.\lambda^{l_1} x.f@^{l_3}x \\ \text{ in } app@(app@\lambda^{l_2} y.y)@\text{True}\end{array}$$

Analysing this expression, we first find that f can be bound to l_2 in the application $app@\lambda^{l_2}y.y$. The result of this application is l_1 which is passed as argument to *app*. At this point it would be unsound to destructively update the binding for f to l_1 since there is still a live occurrence of f bound to l_2. It is easy to see that both l_1 *and* l_2 will eventually be applied to True at $@^{l_3}$.

□

7.1.1 Pairs

The easiest way to add pairs is to treat them like functions:

$$\begin{array}{rcl} C[\![(e_1,e_2)^l]\!]\rho & = & (\{l\},\rho) \\ C[\![\text{let}^l\ (x,y)\ \text{be}\ e_1\ \text{in}\ e_2]\!]\rho & = & \text{let } (\{l_1,\cdots,l_n\},\rho') = C[\![e_1]\!]\rho \\ & & \text{let } (L_i,\rho_i) = C[\![\mathit{fstof}\,(l_i)]\!]\rho' \\ & & \text{let } (L_i',\rho_i') = C[\![\mathit{sndof}\,(l_i)]\!]\rho' \\ & & \text{let } \rho' = \rho \cup [\ x \mapsto \bigcup_{i\in\{1,\cdots,n\}} L_i', \\ & & \qquad\qquad y \mapsto \bigcup_{i\in\{1,\cdots,n\}} L_i''] \\ & & \text{let } \rho'' = \bigcup_{i\in\{1,\cdots,n\}} (\rho_i \cup \rho_i') \\ & & \text{let } (L',\rho''') = C[\![e_2]\!]\rho'' \\ & & \text{in } (L',\rho''') \end{array}$$

This entails that the components of pairs are re-analysed every time they are used. Intuitively, this reflects a lazy (call-by-name) treatment of pairs. The strategy will be safe under any evaluation but is unnecessarily time consuming. Since pairs are not a binding construct, they can be analysed several times under the same assumptions — if the assumptions change, this will be due to an enclosing lambda which will force re-analysis itself.

A better strategy thus uses an extra environment that maps pair-labels to pairs of sets of labels (describing the values that the components can evaluate to). This is exactly the role ϕ plays for pairs in Sestoft's analysis. But while a "universal" environment for functions means loss of information compared to Shivers' analysis, this is not the case for pairs (because they are not binding constructs).

7.2 nCFA

Shivers' flow analysis is an abstraction of an instrumented semantics. This semantics is defined using *contours* (which correspond to *frames*) that keep track of different live instances of variables. Obviously, there is no upper bound on the number of contours during an execution of a program — on the other hand, this is the only infiniteness during execution and thus the target for abstraction.

The semantics allocate a new contour at every application — in the exact instrumented semantics the new contour is picked from an infinite set of contours. In 0CFA the set of contours is a singleton set. Thus every application uses the same contour — this allows us to dispose of the contours altogether and arrive at the analysis presented in the previous section.

In 1CFA a less brutal abstraction is used: there is exactly one contour for each call-site in the program. Thus, if two call-sites apply the same lambda, the bound variable will live in different contours and we will not have to take the union of bindings. E.g. in

$$\begin{array}{l}\text{let } id = \lambda x.x \\ \text{in } \cdots id@^{l_1}\text{True}^{l_2} \cdots id@^{l_3}\text{False}^{l_4} \cdots\end{array}$$

the two bindings to x will not be mixed up, since the first will exist in the contour named l_1 and the second in the contour named l_3.

The memory of 1CFA is short lived, however: consider

$$\begin{array}{l}\text{let } id = \lambda x.x \\ \text{in } \text{ let } f = \lambda y.id@^{l_5}y \\ \qquad \text{in } \cdots f@^{l_1}\text{True}^{l_2} \cdots f@^{l_3}\text{False}^{l_4} \cdots\end{array}$$

The two calls to f are in separate contours, so the bindings for y will live in the contour named l_1 and l_3 resp. The binding for x will live in one contour, namely, the one named l_5. Thus the abstract result of the first application of f will be $\{l_2\}$ but the second will be the set $\{l_2, l_4\}$.

The generalisation nCFA makes the set of contours isomorphic to $Call^n$ where $Call$ is the set of call-sites. Thus 2CFA would be able to handle the above example since the two instances of x would live in the contour named $l_1 \times l_5$ and $l_3 \times l_5$ resp.

nCFA has some of the same flavour as polymorphic flow analysis as presented in chapter 5. There are, however, significant differences:

1. In nCFA, the precision of the first analysed call is better than later analysed.

2. nCFA allows a separate treatment (different contours) for *all* abstraction whereas polymorphic analyses only are able to do this for let- and fix-bound functions. In this sense, nCFA resembles System F based analysis more (discussed in section 8.5).

3. Polymorphic analyses are *not* restricted by the length of the chain of calls and is in this sense ∞CFA (see also the discussion on Jaganathan and Wright's concept of polymorphic splitting in section 11.1 [JW96b]).

7.3 CPS vs. direct style

As mentioned in the introduction to this chapter, Shivers defined his analysis for a CPS-based language. The advantage of CPS-transforming a program is that the resulting program is *tail-recursive* — this is also the crucial prerequisite for Shivers' definition to work: CPS is just a well-know and often used method of transformation of programs to tail-recursive form.

Define tail-recursive terms by the following syntax:

$$\begin{array}{lcl} e & ::= & v@^l v \mid v@^l(v', v'') \mid \text{if}^l\ v \text{ then } v' \text{ else } v' \mid \text{let } x = v \text{ in } v' \mid v \\ v & ::= & \lambda^l x.e \mid \lambda^l(x, c).e \mid \text{True}^l \mid \text{False}^l \mid x \mid \text{fix}^l(f, x).e \mid \text{fix}^l(f, x, c).e \end{array}$$

(where we again for simplicity leave out pairs). Abstractions come in two variants (and thus so do applications): the usual variant with one argument and a variant that takes an argument and a continuation. Similarly, 'fix' comes in a variant with two arguments (the name used for recursive calls, and the argument) and in a variant with an extra continuation argument.

This allows us to simplify the algorithm for 0CFA given in figure 7.1 considerably. The result is shown in figure 7.2.

The new selector *cvarof* picks out the continuation variable of an abstraction or fix, and *fixvarof* picks out the recursion variable of a fix.

Note how the tail-recursiveness has spread to the definition of the analysis: there is no need to return the updated environment. Thus this version of 0CFA which takes advantage of tail-recursiveness will be considerably faster than the general algorithm. The version given in figure 7.2 is identical except for small differences in language to the algorithm given by Shivers.

The improvement of complexity is not for free: applying the analysis of figure 7.1 to a term e can potentially give better results than the above analysis to the result of CPS.

Recall the expression of example 7.3:

$$\begin{array}{l} \text{let } id = \lambda x.x \\ \text{ in } \ \text{if } id@\text{True}^{l_1} \\ \qquad \text{then } id@\text{False}^{l_2} \\ \qquad \text{else True}^{l_3} \end{array}$$

$$
\begin{array}{rcl}
C_v[\![x]\!]\rho & = & \rho(x) \\
C_v[\![\lambda^l x.e]\!]\rho & = & \{l\} \\
C_v[\![\lambda^l (x,c).e]\!]\rho & = & \{l\} \\
C_v[\![\mathrm{True}^l]\!]\rho & = & \{l\} \\
C_v[\![\mathrm{False}^l]\!]\rho & = & \{l\} \\
C_v[\![\mathrm{fix}^l (f,x).v]\!]\rho & = & \{l\} \\
C_v[\![\mathrm{fix}^l (f,x,c).v]\!]\rho & = & \{l\}
\end{array}
$$

$$
\begin{array}{rcl}
C_e[\![\mathrm{if}^l\ v_1\ \mathrm{then}\ v_2\ \mathrm{else}\ v_3]\!]\rho & = & \mathrm{let}\ L_1 = C_v[\![e_1]\!]\rho \\
 & & \mathrm{let}\ L_2 = C_v[\![e_2]\!]\rho \\
 & & \mathrm{let}\ L_3 = C_v[\![e_3]\!]\rho \\
 & & \mathrm{in}\ L_2 \cup L_3 \\
C_e[\![\mathrm{let}\ x = v_1\ \mathrm{in}\ v_2]\!]\rho & = & \mathrm{let}\ L_1 = C_v[\![v_1]\!]\rho \\
 & & \mathrm{let}\ L_2 = C_v[\![v_2]\!](\rho \cup [x \mapsto L_1]) \\
 & & \mathrm{in}\ L_2 \\
C_e[\![v_1 @^l v_2]\!]\rho & = & \mathrm{let}\ \{l_1, \cdots, l_n\} = C_v[\![v_1]\!]\rho \\
 & & \mathrm{let}\ L_2 = C_v[\![v_2]\!]\rho \\
 & & \mathrm{let}\ L_i = \ \mathrm{if}\ l_i\ \mathrm{is\ a\ lambda\ label} \\
 & & \qquad \mathrm{then}\ \ C_e[\![\mathit{bodyof}\,(l_i)]\!] \\
 & & \qquad\qquad (\rho \cup [\mathit{varof}\,(l_i) \mapsto L_2,]) \\
 & & \qquad \mathrm{else}\ \ C_e[\![\mathit{bodyof}\,(l_i)]\!] \\
 & & \qquad\qquad (\rho \cup [\mathit{fixvarof}\,(l_i) \mapsto \{l_i\}, \\
 & & \qquad\qquad\quad \mathit{varof}\,(l_i) \mapsto L_2]) \\
 & & \mathrm{in}\ \bigcup_{i \in \{1,\cdots,n\}} L_i \\
C_e[\![v_1 @^l (v_2, v_3)]\!]\rho & = & \mathrm{let}\ \{l_1, \cdots, l_n\} = C_v[\![v_1]\!]\rho \\
 & & \mathrm{let}\ L_2 = C_v[\![v_2]\!]\rho \\
 & & \mathrm{let}\ L_3 = C_v[\![v_3]\!]\rho \\
 & & \mathrm{let}\ L_i = \ \mathrm{if}\ l_i\ \mathrm{is\ a\ lambda\ label} \\
 & & \qquad \mathrm{then}\ \ C_e[\![\mathit{bodyof}\,(l_i)]\!] \\
 & & \qquad\qquad (\rho \cup [\mathit{varof}\,(l_i) \mapsto L_2, \\
 & & \qquad\qquad\quad \mathit{cvarof}\,(l_i) \mapsto L_3]) \\
 & & \qquad \mathrm{else}\ \ C_e[\![\mathit{bodyof}\,(l_i)]\!] \\
 & & \qquad\qquad (\rho \cup [\mathit{fixvarof}\,(l_i) \mapsto \{l_i\}, \\
 & & \qquad\qquad\quad \mathit{varof}\,(l_i) \mapsto L_2, \\
 & & \qquad\qquad\quad \mathit{cvarof}\,(l_i) \mapsto L_3]) \\
 & & \mathrm{in}\ \bigcup_{i \in \{1,\cdots,n\}} L_i
\end{array}
$$

Figure 7.2: 0CFA for a tail-recursive language

CPS-converting this expression results in

$$\begin{array}{l}\text{let } id = \lambda(x, c).c@x \\ \text{ in } id@(\text{True}^{l_1}, \lambda y.id@(\text{False}^{l_2}, \lambda z.\ \text{if } y \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{then } z \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else True}^{l_3}))\end{array}$$

Analysis of this term starts by updating the binding for x to $\{l_1\}$ and proceeds to the body of id. Here y will be bound to $\{l_1\}$ and id will be applied again. This time x is updated to $\{l_1, l_2\}$ and hence z will be bound to $\{l_1, l_2\}$. So while the analysis can tell us that the conditional (in the transformed case the variable y) can only be l_1 which is better than closure analysis, the analysis will find that the first branch (the variable z) can be either l_1 or l_2 which is worse than the result for the direct style expression. The result of the whole expression will for the CPS-version be $\{l_1, l_2, l_3\}$ in contrast to just $\{l_2, l_3\}$ in direct style. The intuition is that no analysis can be performed in parallel since the program has been sequentialised. Note, however, that 0CFA is still as precise or more precise than closure analysis.

For nCFA the loss is even worse, as CPS-transformation inserts many new applications, thus making remembering the last n calls more "local".

Part III

Extensions and Applications

Chapter 8

Extensions

In this chapter, we will describe a number of extensions of the previously defined flow analyses. Some will be applicable to all analyses while others are extensions of specific systems.

The extensions are not described with the same rigour as the analyses we have presented so far. Thus the present chapter is to a certain extent suggestions for future work rather than presentations of full-fledged analyses.

8.1 Reachability

The analyses presented are able to tell us which values can flow to which destructors. In particular, it can tell us exactly which booleans can flow to a particular conditional. We can exploit this fact, if it turns out that all booleans that can flow to some conditional are True (or False). We can add the following inference rules to any of the type based systems:

$$\text{Bool-E}\ \frac{C; A \vdash^s e : \text{Bool}^L \quad C; A \vdash^s e' : \kappa \quad \forall l \in L.\text{ExpOf}(l) = \text{True}}{C; A \vdash^s \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

$$\text{Bool-E}\ \frac{C; A \vdash^s e : \text{Bool}^L \quad C; A \vdash^s e'' : \kappa \quad \forall l \in L.\text{ExpOf}(l) = \text{False}}{C; A \vdash^s \text{if}^l\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

(retaining the general Bool-E rule).

This idea was also examined by Ayers who called this additional precision reachability [Aye92]. He reported that in practice few expressions were deemed unreachable (that is, will never be evaluated such as e'' when all conditionals are True), but that the additional power in his particular analysis practically came for free (his analysis was equivalent to the analyses presented in chapter 3).

With these rules, the we get invariance under reduction and expansion

of the following rules:

$$(\delta\text{-if}) \quad \begin{array}{l} \text{if True then } e \text{ else } e' \longrightarrow_s e \\ \text{if False then } e \text{ else } e' \longrightarrow_s e' \end{array}$$

(note that makes 'either' redundant in the non-standard reduction system of chapter 6, so this construct as well as its context-propagation rules can be left out).

8.1.1 Reachability and Intersection Flow Analysis

Since the reachability rules allow invariance under reduction and expansion under the standard reduction rules for 'if', it is tempting to state that combining reachability with intersection flow analysis gives us an *exact* analysis — this is, however, as we will see, not true.

We first show an example illustrating that reachability interacts well with intersection types and can give exact results:

Example 8.1 Consider the following expression

$$\begin{array}{l} \text{let } neg = \lambda^{l_5} x.\text{if } x \text{ then False}^{l_3} \text{ else True}^{l_4} \\ \text{in } (neg@\text{True}^{l_1}, neg@\text{False}^{l_2}) \end{array}$$

with intersection types and reachability, we can give neg the type $(\text{Bool}^{\{l_1\}} \rightarrow^{\{l_5\}} \text{Bool}^{\{l_3\}}) \wedge (\text{Bool}^{\{l_2\}} \rightarrow^{\{l_5\}} \text{Bool}^{\{l_4\}})$ and thus find an exact description of the resulting pair.

□

Unfortunately the above (Bool-E) rules are only applicable if $\text{ExpOf}(l)$ is well defined. Thus they will not help us if the conditional can evaluate to a value which is not a boolean (such as a free variable or an application of a free variable).

Example 8.2 Consider the following expression

$$(\lambda^{l_1} z. \left(\begin{array}{l} \text{if } b \\ \text{then if } b \text{ then } z \text{ else True}^{l_2} \\ \text{else if } b \text{ then False}^{l_3} \text{ else } z \end{array} \right))@^{l_4}\text{True}^{l_5}$$

In this expression the reachability rules are not applicable since b is a free variable (which we can assume has type $\text{Bool}^{\{l_6\}}$). Even though this expression will reduce to True^{l_5} for any value of b, we are not able to give it a better description than $\text{Bool}^{\{l_2,l_3,l_5\}}$. □

Let e be a given expression. Let a truth-assumption $\mathcal{T}_e$ for e be a map from (boolean) labels $l \notin \mathcal{L}_e$ to sets of truth-values. Given $\mathcal{T}_e$, define (overloaded) predicates True? and False? as follows:

$$\begin{array}{rll} \text{True?}(l) & \text{iff} & l \in \mathcal{L}_e \wedge \text{ExpOf}(l) = \text{True} \\ & \text{or} & l \notin \mathcal{L}_e \wedge \mathcal{T}_e(l) = \text{True} \\ \text{False?}(l) & \text{iff} & l \in \mathcal{L}_e \wedge \text{ExpOf}(l) = \text{False} \\ & \text{or} & l \notin \mathcal{L}_e \wedge \mathcal{T}_e(l) = \text{False} \\ \text{True?}(L) & \text{iff} & \forall l \in L : \text{True?}(l) \\ \text{False?}(L) & \text{iff} & \forall l \in L : \text{False?}(l) \end{array}$$

Now the new type rules can be based on the True? and False? predicates:

$$\text{Bool-E}\ \frac{A \vdash^{\wedge} e : \text{Bool}^{L} \quad A \vdash^{\wedge} e' : \kappa \quad \text{True?}(L)}{A \vdash^{\wedge} \text{if}^{l}\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

$$\text{Bool-E}\ \frac{A \vdash^{\wedge} e : \text{Bool}^{L} \quad A \vdash^{\wedge} e'' : \kappa \quad \text{False?}(L)}{A \vdash^{\wedge} \text{if}^{l}\ e\ \text{then}\ e'\ \text{else}\ e'' : \kappa}$$

Given any truth-assumption $\mathcal{T}_e$ we are now able to do flow inference: we can use the notation $A \vdash^{\wedge}_{\mathcal{T}_e} e : \kappa$ for $A \vdash^{\wedge} e : \kappa$ under assumption $\mathcal{T}_e$. Given A and κ call a set $\mathcal{TS}$ of truth-assumptions *complete* if

1. $l_1, \cdots, l_m, l_{m+1}, \cdots, l_n$ are all labels in $A \vdash^{\wedge} e : \kappa$ such that $l_i \notin \mathcal{L}_e$
2. For all $\langle b_1, \cdots, b_m \rangle \in \{\langle \{t_1\}, \cdots, \{t_m\} \rangle \mid t_i \in \{\text{True}, \text{False}\}\}$ there exists $\mathcal{T} \in \mathcal{TS}$ such that $\langle \mathcal{T}(l_1), \cdots, \mathcal{T}(l_m) \rangle = \langle b_1, \cdots, b_m \rangle$, and
3. $\mathcal{T}(l_i) = \{\text{True}, \text{False}\}$ for all $\mathcal{T} \in \mathcal{TS}$ and $m < i \leq n$

Intuitively, l_i where $m < i \leq n$ are the labels for which we make no assumptions and for l_j where $1 \leq j \leq m$ we need to check that we can find the same result for *all* combinations of assumptions.

We define $A \vdash^{\wedge\text{-reach}} e : \kappa$ to be true if there exists $\mathcal{TS}$ such that

1. $\mathcal{TS}$ is a complete set of truth-assumptions w.r.t. A and κ, and
2. $A \vdash^{\wedge}_{\mathcal{T}} e : \kappa$ for all $\mathcal{T} \in \mathcal{TS}$

Example 8.3 Recall example 8.2. By proving

$$b : \text{Bool}^{\{l_6\}} \vdash^{\wedge}_{[l_6 \mapsto \{\text{True}\}]} (\lambda^{l_1} z. \left(\begin{array}{l} \text{if } b \\ \text{then if } b \text{ then } z \text{ else True}^{l_2} \\ \text{else if } b \text{ then False}^{l_3} \text{ else } z \end{array} \right) @^{l_4} \text{True}^{l_5} : \text{Bool}^{\{l_5\}}$$

and

$$b : \text{Bool}^{\{l_6\}} \vdash^{\wedge}_{[l_6 \mapsto \{\text{False}\}]} (\lambda^{l_1} z. \left(\begin{array}{l} \text{if } b \\ \text{then if } b \text{ then } z \text{ else True}^{l_2} \\ \text{else if } b \text{ then False}^{l_3} \text{ else } z \end{array} \right) @^{l_4} \text{True}^{l_5} : \text{Bool}^{\{l_5\}}$$

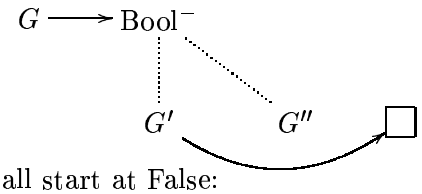
we conclude

$$b : \mathrm{Bool}^{\{l_6\}} \vdash^{\wedge\text{-reach}} (\lambda^{l_1} z. \left(\begin{array}{l} \text{if } b \\ \text{then if } b \text{ then } z \text{ else } \mathrm{True}^{l_2} \\ \text{else if } b \text{ then } \mathrm{False}^{l_3} \text{ else } z \end{array} \right) @^{l_4} \mathrm{True}^{l_5} : \mathrm{Bool}^{\{l_5\}}$$

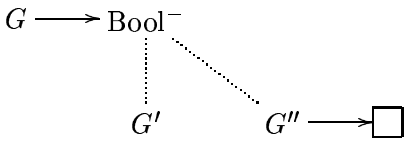
□

8.1.2 Reachability in Graphs

A similar idea is applicable with the graph based analyses. If all Bool^+-Bool^- paths ending at a particular $(\mathrm{Bool}^-)^l$ node start at True^1 then we can "cut" an edge in the graph:



and similarly if they all start at False:



Notice that applying such a transformation can trigger the applicability of the same rule at other Bool^- nodes. Exactly the same rule is applicable in typed graphs.[2]

If there are paths starting at a free variable and ending at $(\mathrm{Bool}^-)^l$, we can do as in the type system: only if the condition for "cutting" is applicable under *all* assignments of truth-values to free variables, can we "cut".

8.2 Union Types

The idea employed in section 8.1.1 is a special case of *union types*. To obtain a union based flow analysis we add the following rules to the intersection based analysis of chapter 6:

Semi logical rules:

$$\vee\text{-E}\ \frac{A, x : \kappa_1 \vdash^{\wedge\vee} e : \kappa \quad A, x : \kappa_2 \vdash^{\wedge\vee} e : \kappa \quad A \vdash^{\wedge\vee} e' : \kappa_1 \vee \kappa_2}{A \vdash^{\wedge\vee} e[e'/x] : \kappa}$$

[1] to be exact $\mathrm{ExpOf}(l)$ is True for all l such that there is a path starting from $(\mathrm{Bool}^+)^l$. This would be straightforward to express without reference to labels if we used 'True' and 'False' nodes instead of Bool^+.

[2] In untyped graphs, we have to remove edges added by closing rules triggered by a "cut" edge. This is avoided in typed graphs.

Logical rules: [3]

$$\frac{}{\vdash^{\wedge\vee} \kappa \leq \kappa \vee \kappa'} \qquad \frac{}{\vdash^{\wedge\vee} \kappa' \leq \kappa \vee \kappa'}$$

$$\frac{\vdash^{\wedge\vee} \kappa_1 \leq \kappa \quad \vdash^{\wedge\vee} \kappa_2 \leq \kappa}{\vdash^{\wedge\vee} \kappa_1 \vee \kappa_2 \leq \kappa}$$

Type-specific logical rules:

$$\frac{}{\vdash^{\wedge\vee} (\kappa_1 \to^L \kappa_2) \wedge (\kappa_1' \to^L \kappa_2) \leq \kappa_1 \vee \kappa_1' \to^L \kappa_2}$$

The strong reachability flow analysis of section 8.1.1 can be seen as a special case of this in the following manner. Instead of having general boolean types we introduce types True and False and replace the boolean standard typing rules with

$$\text{Bool-I } \frac{}{A \vdash \text{True} : \text{True}} \qquad \frac{}{A \vdash \text{False} : \text{False}}$$

$$\text{Bool-E } \frac{A \vdash e : \text{True} \quad A \vdash e' : t}{A \vdash \text{if } e \text{ then } e' \text{ else } e'' : t} \qquad \frac{A \vdash e : \text{False} \quad A \vdash e'' : t}{A \vdash \text{if } e \text{ then } e' \text{ else } e'' : t}$$

and annotating types True and False in the obvious fashion. Notice that the "standard" rule for 'if' is derivable from these rules using the ($\vee$-E) rule

$$\frac{A \vdash^{\wedge\vee} e : \text{True}^L \vee \text{False}^L \quad A \vdash^{\wedge\vee} e' : \kappa \quad A \vdash^{\wedge\vee} e'' : \kappa}{A \vdash^{\wedge\vee} \text{if}^l \ e \text{ then } e' \text{ else } e'' : \kappa}$$

We are now able to do flow inference under any assumption on the truth-type of free variables and by using the $\vee$-E rule to combine inference trees to the more liberal True $\vee$ False assumption.

We conjecture that general union types do *not* give any additional power over the strong reachability analysis of section 8.1.1. We suggest that a proof of this can be given as a proof transformation "sifting" all applications of the $\vee$-E rule to the root of the inference tree.

Thus, it would be superfluous to add general union types, but not only that: we would lose invariance under beta-reduction as the following example shows:

[3] Barbenera et al. use an equivalent but slightly less elegant formulation [BDCd95]. Using their system as inspiration would lead to:

$$\frac{}{\vdash^{\wedge\vee} \kappa \vee \kappa \leq \kappa} \qquad \frac{}{\vdash^{\wedge\vee} \kappa \leq \kappa \vee \kappa'} \qquad \frac{}{\vdash^{\wedge\vee} \kappa' \leq \kappa \vee \kappa'}$$

$$\frac{\vdash^{\wedge\vee} \kappa_1 \leq \kappa_1' \quad \vdash^{\wedge\vee} \kappa_2 \leq \kappa_2'}{\vdash^{\wedge\vee} \kappa_1 \vee \kappa_2 \leq \kappa_1' \vee \kappa_2'}$$

Example 8.4 Consider the term

$$\lambda^{l_1}x.\lambda^{l_2}y.\lambda^{l_3}z.(\lambda^{l_5}f.x@f@f)@((\lambda^{l_4}t.t)@y@z)$$

This can reduce to:

$$\lambda^{l_1}x.\lambda^{l_2}y.\lambda^{l_3}z.x@((\lambda^{l_4}t.t)@y@z)@((\lambda^{l_4}t.t)@y@z) \tag{8.1}$$

(which is the term that we are really interested in — the first was only to show how we got the same label twice). This can reduce to either

$$\lambda^{l_1}x.\lambda^{l_2}y.\lambda^{l_3}z.x@(y@z)@((\lambda^{l_4}t.t)@y@z) \tag{8.2}$$

or

$$\lambda^{l_1}x.\lambda^{l_2}y.\lambda^{l_3}z.x@((\lambda^{l_4}t.t)@y@z)@(y@z) \tag{8.3}$$

Both expressions 8.2 and 8.3 can reduce to

$$\lambda^{l_1}x.\lambda^{l_2}y.\lambda^{l_3}z.x@(y@z)@(y@z) \tag{8.4}$$

We can give expressions 8.1 and 8.4 type (where we leave out unimportant annotations)

$$\begin{array}{l}(\mathrm{Bool}^{\{l_5\}} \to \mathrm{Bool}^{\{l_5\}} \to \mathrm{Bool}^{\{l_8\}}) \wedge (\mathrm{Bool}^{\{l_6\}} \to \mathrm{Bool}^{\{l_6\}} \to \mathrm{Bool}^{\{l_8\}}) \\ \quad \to^{\{l_1\}} (\mathrm{Bool}^{\{l_7\}} \to \mathrm{Bool}^{\{l_5\}} \vee \mathrm{Bool}^{\{l_6\}}) \to^{\{l_2\}} \mathrm{Bool}^{\{l_7\}} \to^{\{l_3\}} \mathrm{Bool}^{\{l_8\}}\end{array}$$

but this type can be given to neither expressions 8.2 nor 8.3. This is because the intermediate forms lose syntactical equivalence between the two instances of f. We can conclude that types are preserved neither by beta reduction nor expansion. This example is adopted from an example by Pierce [Pie91].

□

It can be shown that a standard union type system is invariant under *parallel* beta reduction and expansion [BDCd95] and we believe that this result can be adopted to union based flow analysis.

Observation 8.5 *The intersection and union based strictness analysis of Jensen section [Jen92] includes the following* $\vee$ *elimination rule*

$$\frac{A, x : \kappa \vdash e : \kappa'' \quad A, x : \kappa' \vdash e : \kappa''}{A, x : \kappa \vee \kappa' \vdash e : \kappa''}$$

which is the rule from the sequent calculus formulation of union types instead of the natural deduction $\vee$*-E rule given above.*

Without the cut rule of sequent calculus systems, Jensen's rule is strictly weaker than the natural deduction rule. This implies that his system must lack invariance under beta-reduction in a "worse" way than necessary (the full version is, as noted above, invariant under parallel *beta reduction and expansion).*

This manifests itself in e.g. reducing let $x = e$ in e' *to* $e'[e/x]$ *where the sequent calculus rule is applicable to* e *in the redex but not in the reduct where* e *is not bound to any variable.*

8.3 Usedness

In the analyses presented, we have made a virtue of being sound under any reduction order. This has the advantage of allowing hyper-strict evaluation (such as partial evaluation) and compiler optimisations based on beta-reduction.

If we are interested in a fixed evaluation strategy only, the general approach will predict redexes that will never arise during evaluation. The information about whether a subexpression is needed or not, can be inferred in a separate analysis and the flow analysis be changed to take the information into account. Instead of using a separate analysis, it is possible to build this form of "dead code elimination" into the flow analysis.

8.3.1 Lazy Evaluation

We might expect that expressions that never are used do not contribute to the flow information. Consider the following expression, however:

$$(\lambda x.f@\text{True})@(f@\text{False})$$

Our analyses will find that f can be applied to both True and False, but under lazy evaluation (f@False) will never be evaluated, and we might be interested in realising that f is never applied to False.

Our first option is to infer usedness by a separate analysis and let the flow analysis use the information. The usedness analysis could give unused expressions a special type Ω. The only difference would be when computing the flow function, no flow inferred for subexpressions given type Ω should be included.

The second option is to build the usedness analysis into the flow analysis by adding the following:

Formulae:

$$\Omega\ \frac{}{\Omega \in \mathcal{K}(t)}$$

Semi-logical rules:

$$\Omega\ \frac{}{C; A \vdash e : \Omega}$$

and require that if $C; A \vdash e : \kappa$ is the final judgement then neither A nor κ contains Ω. This also handles lazy pairs, since the components of a pair that is destructed but where the components are not used, can be given type $\Omega \times \Omega$.

8.3.2 Eager Evaluation

For call-by-value we might wish similar optimisations for terms such as

$$(\lambda x.f@\text{True})@(\lambda^{l_1} z.f@\text{False})$$

where the fact that the body of the second lambda is not evaluated implies that f will never be applied to False.

We have the same two options as in the lazy case. The only change to the built-in version is to require that no argument in an application has type Ω and that we do not build pairs with components of type Ω:

$$\rightarrow\text{-E}\ \frac{C; A \vdash^s e : \kappa' \rightarrow^\ell \kappa \quad C; A \vdash^s e' : \kappa' \quad \kappa' \neq \Omega}{C; A \vdash^s e@^l e' : \kappa}$$

$$\times\text{-I}\ \frac{C; A \vdash^s e : \kappa \quad C; A \vdash^s e' : \kappa' \quad C \vdash^s \{l\} \subseteq \ell \quad \kappa \neq \Omega \quad \kappa' \neq \Omega}{A \vdash^s (e, e')^l : \kappa \times^\ell \kappa'}$$

The first rule reflects the fact that all arguments are evaluated. The argument type can contain Ω types as in the above example where $\lambda z.f@\text{False}$ can be given type $\Omega \rightarrow^{\{l_1\}} \Omega$.

The second rule reflects that if a pair is ever used then its components are evaluated.

8.4 Simple Polymorphism

In chapter 5 we added polymorphism (ML- and fix-polymorphism) to the subtype based flow analysis of chapter 3. There is, however, no reason why we cannot add polymorphism to the simply typed system of chapter 2.

As this will be strictly weaker than the subtype polymorphic analyses, the aim should be to improve the complexity. We will not go into very much detail about this system, just note that we have principal types due to an argument similar to the arguments in chapter 5 using the following properties:

1. Analysing an expression e results in a constraint set which is linear in the size of the untyped term: namely $\{l\} \subseteq \alpha$ for each constructor annotated with l.

2. Since the set of constraints is fixed in this manner, the Kleene-Mycroft sequences for simple polymorphism must have the property that $\sigma_{i+1} = close_A(S(C_i \Rightarrow \kappa_i))$ where S unifies variables bound by σ_i. Thus the number of variables in a Kleene-Mycroft sequence will be strictly decreasing until a fixed point is found. This bounds the length of Kleene-Mycroft sequences by the size of the program and the computation of S must be linear as well (checking for $\sqsubseteq$ will depend only on whether S is the identity or not).

This suggests that polymorphically recursive, simple flow analysis is computable in cubic time using the accelerated algorithm of subsection 5.3.5. If we can use the ideas for unification employed by efficient implementations of simple flow analysis, this complexity might even be improved.

The above argument is enchanting, but more rigour is required before the cubic (or possibly even quadratic)complexity (can be promoted to more than a conjecture.

8.5 Other Inference-Based Analyses

System F is a powerful generalisation of ML- and fix-polymorphism[4]. The extension consists of not restricting the use of polymorphism — in other words higher order and first order formulae are not distinguished.

It is straightforward to define a System F based flow analysis, but such an analysis is likely to be too complex for practical use. Rank n fragments could, however, be more promising.

Along a similar line, restrictions such as rank n of intersection based analysis could be interesting for achieving precise but practical analyses. Independently of this work, Banerjee investigated rank 2 intersection types as a basis for flow analysis. He gives an algorithm for his analysis but does not give any estimate on its complexity (for further description and comparison with this work, see chapter 11).

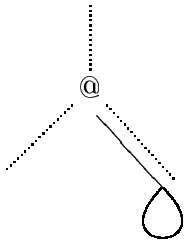
8.6 Labelled Graphs

The imprecision of graph based analysis can be seen to be due to the graph containing unrealisable paths. We can hope to improve the analysis by introducing criteria for realisable paths. Such a criterion is known and given by Asperti who refines his notion of well-balanced paths (see section 4.5) to the notion of legal paths. We follow [AG96] closely in the presentation.

Recall that wbp's corresponded to our paths (in unlabeled graphs) except that they went in the opposite direction.

Definition 8.6 *Let ϕ be a wbp. An* elementary @-cycle *of ϕ is a subpath ψ starting from and ending by going through the argument edge of an application node and internal to the argument e of the application (i.e. not traversing any variables which are free in e).*

[4]Historically, System F was introduced independently by Girard [Gir72] and Reynolds [Rey74]

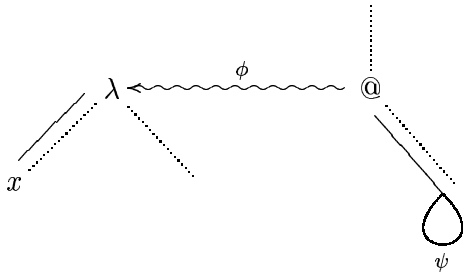


Definition 8.7 *The concepts of @-cycle and v-cycle (cycle over a variable) are defined by mutual induction:*

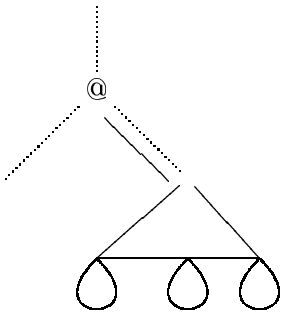
Base case: *Every elementary @-cycle is a @-cycle.*

Induction case:

v-cycle: Every cyclic path of the form $x\lambda(\phi)^r @\psi @\phi\lambda x$ *where* ϕ *is a wbp and* ψ *is a @-cycle is a v-cycle.*



@-cycle: Every path ψ *starting and ending at the argument edge of an application node and composed of sub-paths internal to the argument e with v-cycles over free variables of e is a @-cycle.*



Proposition 8.8 (Corollary 5.1.19 of [AG96]) *Let ϕ be a wbp with a @-cycle @ψ@. Then ϕ can be uniquely decomposed as:*

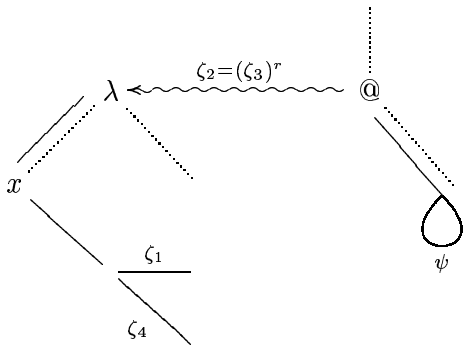
$$\zeta_1 x \lambda \zeta_2 @ \psi @ (\zeta_3)^r \lambda y \zeta_4$$

where both ζ_2 and ζ_3 are wbp's.

Call ζ_2 and ζ_3 the *call* resp. *return* path of the @-cycle ψ. Call the last edge of ζ_1 and the first edge of ζ_4 the *discriminants* of the call resp. return paths.

Definition 8.9 *A wbp is a* legal *path if and only if the call and return paths of any @-cycle are the reverse of the other and their discriminants are equal.*

Note that the condition also makes the variables x and y of proposition 8.8 equal. With the notation of proposition 8.8, we can draw a cycle on a legal path as follows:



Asperti proved legal paths equivalent to *consistent* paths [Lam90] and *regular* paths [Gir89] (the proof can be found in [ADLR94]). The aim of these different kinds of paths was to specify the redexes in an expression: to obtain optimal reduction, no such redex may be copied [Lév80].

Legal paths give an exact characterisation of the possible redexes during reduction and we anticipate that if the notion was extended to our full language, we would find the analysis equivalent to intersection based analysis.

Statman's lemma implies that any nontrivial predicate on simply typed lambda terms that is preserved under beta reduction and expansion is non-elementary recursive [Sta79, Mai92]. Any search of a useful algorithm implementing the above is thus in vain (if it should be useful).

Furthermore, the notion of legal paths is not a very good starting point for the development of practical algorithms. Inspired by the work by Horowitz, Reps and Sagiv we could hope to label edges in the graph and to characterise realisable (legal) paths using a language over these labels [RHS95, RSH94, SRH95]. By Statman's lemma, this language cannot be

context-free as this would lead to polynomial time algorithm using context-free reachability [Yan90] (like Horowitz, Reps and Sagiv) but we could hope to find languages which had non-trivial, context-free abstractions.

Horowitz, Reps and Sagiv's idea is to label every function call with an opening parenthesis '$(_n$' and every return from a function with a closing parenthesis '$)_n$'. Different call/return pairs are given different numbers n and realisable paths are now paths that traverse call/return labels in a well-balanced manner.

For typed graphs, the natural generalisation is to label edges in variable occurrence cables: backwards edges are labelled with a left parenthesis (since values flowing through such a cable is an argument) and forwards edges are labelled with a right parenthesis. The subscript n corresponds to different occurrences of the variable. (We note that labels would be needed in the branch cables of conditionals and binding- and result cable of fix as well.)

Simply requiring paths to traverse labels in a well-balanced manner is, however, much too restrictive. The following simple example can illustrate some of the problems

$$\begin{array}{l}\text{let } f = \lambda x.x \\ \text{in } f@(f@\text{True})\end{array}$$

The value True enters f, traverses x, leaves f, enters f again, traverses x and finally leaves f. With the parenthesis notation this could amount to $(^f_1\ (^x_3\ )^f_1\ (^f_2\ (^x_3\ )^f_2$ where we use superscript to indicate the variable occurrence traversed (strictly not necessary).

First, this path is not well-balanced. In general with higher-order functions, call/return are not simply nested as in the first-order case[5]. This problem seems to be solvable by considering well-balancedness for each variable in turn.

Secondly, the two traversals of $(^x_3$ are essentially through two different instances of x. In the first-order case, this does not cause any problems, but examples can be found where this turns out to disallow paths that correspond to potential redexes (we have only been able to find examples of 2-level paths with this problem).

8.7 Shivers' CFA and Linearity

Shivers' nCFA analyses can, with good effect, make use of information about linearity of variables. In the application case, the binding of the formal parameter x is updated to be the union of its previous binding and the argument the abstraction is applied to. This is necessary since there might

[5] As this expression is first-order, it can be handled by Horowitz, Reps and Sagiv — this is because x would not be labelled. We could leave out labels on carrier paths as an optimisation, but it would still be easy to generalise the above example to obtain paths of a similar kind.

be other instances of x that are still alive. Now, assume that x is linearly used: this implies exactly that no other instances of x can be alive at the point where this application takes place. It will thus be safe to *destructively update* the binding for x.

As an example, consider (from example 7.4):

$$\begin{array}{l}\text{let } id = \lambda x.x \\ \text{ in } (\lambda^{l_3} y.id@\text{False}^{l_2})@^{l_4}(id@\text{True}^{l_1})\end{array}$$

We easily realise that x is linear. In the first application of id 0CFA will find that x is bound to $\{l_2\}$. In the second, standard 0CFA would have to update the binding of x to $\{l_2, l_3\}$. If we make use of the fact that x is linear, we will overwrite the binding with $\{l_3\}$ instead.

The optimisation is valid not only if a variable is linear (used exactly once) but if it is affine (used at most once). A suitable analysis for finding affine variables is described by Turner, Wadler and the present author [TWM95]. We leave a proof of validity of this optimisation to future work.

Chapter 9

Other Standard Type Systems

The analyses based on untyped graphs are applicable to untyped languages as well as languages with any type system. In contrast, the type based analyses as well as the analyses based on typed graphs rely heavily on the presence of standard types to guide the analysis.

In this chapter we will try to convince the reader that the flow analyses presented are not restricted to the simple language described in section 1.6. We will discuss both more powerful language constructs and more powerful type systems. The extensions considered are ML polymorphism (section 9.1), sum types (section 9.2), recursive types (section 9.3) and dynamic types (section 9.4).

For illustration, we will show the extensions to the subtype flow analysis of chapter 3 and the typed flow graphs of chapter 4, but hope that it will be clear that the extensions are equally applicable to the other analyses.

We will argue the correctness of the proposed extensions, but will leave a more thorough investigation as future work.

9.1 ML Polymorphism

We have studied polymorphism at the level of annotations. For standard type polymorphism we have the types:

$$\begin{array}{rcl} t & ::= & \tau \mid \mathrm{Bool} \mid t \to t' \mid t \times t' \\ \sigma & ::= & t \mid \forall \vec{\tau}.t \end{array}$$

For convenience, we add explicit syntax for quantification and instantiation

$$e \quad ::= \quad \Lambda \vec{\tau}.e \mid e\{\vec{t}\}$$

The type rules for standard ML polymorphism are given in figure 9.1. It should be no cause of confusion that σ ranges over both standard type schemes and flow schemes.

$$\forall\text{-I}\ \frac{A \vdash e : t}{A \vdash \Lambda\vec{\tau}.e : \forall\vec{\tau}.t} \quad (\vec{\tau} \text{ not free in } A)$$

$$\forall\text{-E}\ \frac{A \vdash e : \forall\vec{\tau}.t'}{A \vdash e\{\vec{t}\} : t'[\vec{t}/\vec{\tau}]}$$

$$\text{let}\ \frac{A \vdash e : \sigma \quad A, x : \sigma \vdash e' : t'}{A \vdash \text{let } x = e \text{ in } e' : t'}$$

Figure 9.1: ML-polymorphism

It is possible to add annotation to quantify and instantiate expressions, allowing us to trace which quantifications are instantiated where. Since this kind of flow is trivial for ML-polymorphism (a quantifier can only flow to the occurrences of the let-bound variable that was quantified) we choose not to annotate $\forall$. If our standard language included more powerful polymorphism, such a generalisation of flow analysis could be useful.

The inference system for flow analysis of ML polymorphic terms is presented in figure 9.2.

Note that ML-polymorphism implies a certain kind of polymorphism at the annotation level. Consider the following expression:

$$\begin{array}{l}\text{let } id = \Lambda\tau.\lambda^{l_3}x.x \\ \text{ in } (id\{\text{Bool}^{\{l_1\}}\}@\text{True}^{l_1}, id\{\text{Bool}^{\alpha} \rightarrow^{\{l_2\}} \text{Bool}^{\alpha}\}@\lambda^{l_2}y.y)\end{array}$$

where we obtain the exact result as a side effect of standard type polymorphism. This kind of polymorphism is, however, not as general as let-polymorphic flow analysis. Consider:

$$\begin{array}{l}\text{let } f = \lambda^{l_2}x.\text{if } x \text{ then } x \text{ else False}^{l_1} \\ \text{ in } \ldots\end{array}$$

which has a monomorphic standard type, but where polymorphism at annotation level can help improve precision.

9.1.1 ML Polymorphism in Typed Graphs

Intuitively, a type τ can carry any value since it might be instantiated to any type. For graphs, however, the opposite intuition is more fruitful: no value is carried along a τ-cable, since, as long as the value has type τ, it *cannot* be used. Thus a τ-cable is no cable at all and the appropriate connections are made at the instantiation node. A $\forall\vec{\tau}.t$ cable is a t-cable.

Formulae:

$$\text{Bool}\ \frac{}{\text{Bool}^{\ell} \in \mathcal{K}^{\leq}(\text{Bool})}$$

$$\to\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \kappa' \in \mathcal{K}^{\leq}(t')}{\kappa \to^{\ell} \kappa' \in \mathcal{K}^{\leq}(t \to t')} \qquad \times\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \kappa' \in \mathcal{K}^{\leq}(t')}{\kappa \times^{\ell} \kappa' \in \mathcal{K}^{\leq}(t \times t')}$$

$$\tau\ \frac{}{\tau \in \mathcal{K}^{\leq}(\tau)} \qquad \forall\ \frac{\kappa \in \mathcal{K}^{\leq}(t)}{\forall\tau.\kappa \in \mathcal{K}^{\leq}(\forall\vec{\tau}.t)}$$

Semi- and non-logical rules

$$\forall\text{-I}\ \frac{C; A \vdash^{\leq} e : \kappa}{C; A \vdash^{\leq} \Lambda\vec{\tau}.e : \forall\vec{\tau}.\kappa} \quad (\vec{\tau} \text{ not free in } A)$$

$$\forall\text{-E}\ \frac{C; A \vdash^{\leq} e : \forall\vec{\tau}.\kappa' \quad \vec{\kappa} \in \mathcal{K}^{\leq}(\vec{t})}{C; A \vdash^{\leq} e\{\vec{t}\} : \kappa'[\vec{\kappa}/\vec{\tau}]}$$

$$\text{let}\ \frac{C; A \vdash^{\leq} e : \kappa \quad C; A, x : \kappa \vdash^{\leq} e' : \kappa'}{C; A \vdash^{\leq} \text{let } x = e \text{ in } e' : \kappa'}$$

Figure 9.2: Flow analysis of ML-polymorphic programs

Since t-cables and $\forall\vec{\tau}.t$-cables are the same, quantification nodes just pass on its incoming cable:

$$\mathcal{TG}(\Lambda\vec{\tau}.e) = \mathcal{TG}(e) \Longrightarrow \boxed{\forall^+} \Longrightarrow \square$$

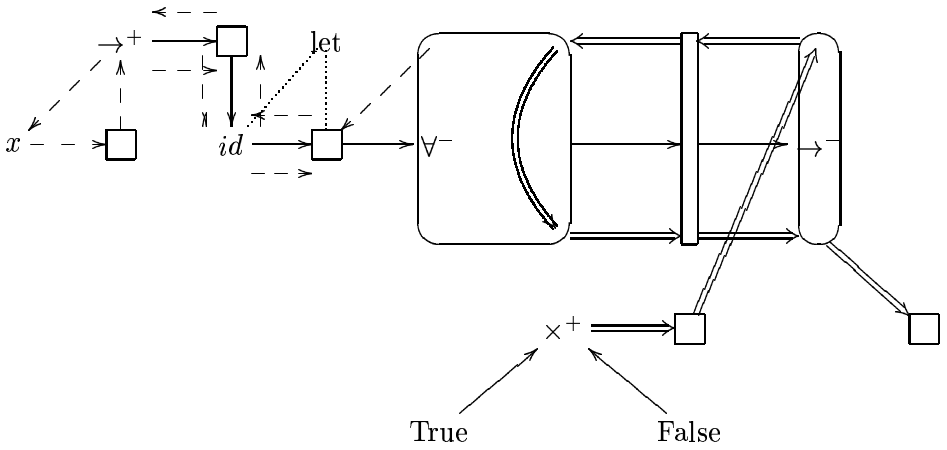
(we can choose to leave out quantification nodes entirely).

At instantiation $t'[\vec{t}/\vec{\tau}]$ all occurrences of the bound variables τ_i (i'th component of $\vec{\tau}$) are replaced by the same type t_i (i'th component of $\vec{t}$). We model this in the instantiation node as follows: if $\tau_i^{(j)}$ is a negative and $\tau_i^{(k)}$ a positive occurrence of τ_i in t' then add a cable in the instantiation node from $t_i^{(j)}$ to $t_i^{(k)}$. All edges in t' are connected to the "same" edges in $t'[\vec{t}/\vec{\tau}]$.

This is best illustrated by an example. Consider

$$\begin{array}{l}\text{let } id = \lambda x.x \\ \text{ in } id@(\text{True}, \text{False})\end{array}$$

where we assume that id is given type $\forall\tau.\tau \to \tau$. The graph for this expression looks as follows:



This approach relies on the same intuition as "Theorems for Free", that a function cannot touch arguments of polymorphic type [Wad89].

The graph contains the $?^+$-$?^-$ paths that we expect, but if we want to know which values a variable can be bound to, this approach is insufficient. The solution is to let τ-cables be single edges (the dashed edges in the graph above) that can carry the top label of the types to which it is instantiated. In the graph, this means that the τ-cables carry the label of the pair around in the let-bound expression — in itself insufficient to carry True and False to the root.

Note, that this approach to standard ML-polymorphism gives the same degree of polymorphism at flow level as the type-based approach: in the example above, the labels of the pair (True, False) reach the root of the application node independently of whether there are any other calls to id.

9.2 Sum Types

In this section we show how our analyses can be extended to handle languages with sum types. Types are as follows

$$t \quad ::= \quad \text{Bool} \mid t \to t' \mid t \times t' \mid t + t' \mid 1$$

Sum types come with associated syntax:

$$e \quad ::= \quad \text{inl}(e) \mid \text{inr}(e) \mid \text{case } e \text{ of inl}(x) \ \mapsto \ e'; \ \text{inr}(y) \ \mapsto \ e'' \mid u$$

where u is the unit of type 1. The type rules are given in figure 9.3.

$$1 \ \frac{}{A \vdash u : 1}$$

$$+\text{-I} \ \frac{A \vdash e : t}{A \vdash \text{inl}(e) : t + t'} \qquad \frac{A \vdash e : t'}{A \vdash \text{inr}(e) : t + t'}$$

$$+\text{-E} \ \frac{A \vdash e : t + t' \quad A, x : t \vdash e' : t'' \quad A, y : t' \vdash e'' : t''}{A \vdash \text{case } e \text{ of inl}(x) \ \mapsto \ e'; \ \text{inr}(y) \ \mapsto \ e'' : t''}$$

Figure 9.3: Sum Types

The new expressions are given a label:

$$e \quad ::= \quad \text{inl}^l(e) \mid \text{inr}^l(e) \mid \text{case}^l \ e \text{ of inl}(x) \ \mapsto \ e'; \ \text{inr}(y) \ \mapsto \ e'' \mid u^l$$

and annotations are added to the singleton type and the sum type constructor. The rules are given in figure 9.4.

Note that we actually *did* have sum types in the original language, as booleans can be considered as the sum $1 + 1$. It is easy to see that with this definition of booleans, we would arrive at rules similar to those we had in figure 3.2. The only difference is that 'True' and 'False' are represented by $\text{inl}(u)$ resp. $\text{inr}(u)$ and will thus each have two labels instead of one (but no increased precision).

9.2.1 Sum Types in Typed Graphs

To extend typed graphs with sum types, we first have to define cables carrying values of sum type:

A $(t + t')$-cable is $\overset{\overset{1}{\Longrightarrow}}{\longrightarrow}\!\!\!\!\!\!\!\!\underset{\underset{2}{\Longrightarrow}}{}$ where $\overset{1}{\Longrightarrow}$ is a t-cable and $\overset{2}{\Longrightarrow}$ is a t'-cable.

Formulae:

$$\text{Bool}\ \frac{}{\text{Bool}^{\ell} \in \mathcal{K}^{\leq}(\text{Bool})}$$

$$\to\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \kappa' \in \mathcal{K}^{\leq}(t')}{\kappa \to^{\ell} \kappa' \in \mathcal{K}^{\leq}(t \to t')} \qquad \times\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \kappa' \in \mathcal{K}^{\leq}(t')}{\kappa \times^{\ell} \kappa' \in \mathcal{K}^{\leq}(t \times t')}$$

$$+\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \kappa' \in \mathcal{K}^{\leq}(t')}{\kappa +^{\ell} \kappa' \in \mathcal{K}^{\leq}(t + t')} \qquad 1\ \frac{}{1^{\ell} \in \mathcal{K}^{\leq}(1)}$$

Type-specific logical rules:

$$\text{Unit}\ \frac{C \vdash \ell_1 \subseteq \ell_2}{C \vdash^{\leq} 1^{\ell_1} \leq 1^{\ell_2}}$$

$$\text{Sum}\ \frac{C \vdash^{\leq} \kappa_1 \leq \kappa_1' \quad C \vdash^{\leq} \kappa_2 \leq \kappa_2' \quad C \vdash \ell_1 \subseteq \ell_2}{C \vdash^{\leq} \kappa_1 +^{\ell_1} \kappa_2 \leq \kappa_1' +^{\ell_2} \kappa_2'}$$

Non-logical rules:

$$\text{1-I}\ \frac{}{C; A \vdash^{\leq} u^{l} : 1^{\{l\}}}$$

$$+\text{-I}\ \frac{C; A \vdash^{\leq} e : \kappa}{C; A \vdash^{\leq} \text{inl}^{l}(e) : \kappa +^{\{l\}} \kappa'} \qquad \frac{C; A \vdash^{\leq} e : \kappa'}{C; A \vdash^{\leq} \text{inr}^{l}(e) : \kappa +^{\{l\}} \kappa'}$$

$$+\text{-E}\ \frac{C; A \vdash^{\leq} e : \kappa +^{\ell} \kappa' \quad A, x : \kappa \vdash^{\leq} e' : \kappa'' \quad A, y : \kappa' \vdash^{\leq} e'' : \kappa''}{C; A \vdash^{\leq} \text{case}^{l}\ e\ \text{of inl}(x)\ \mapsto\ e';\ \text{inr}(y)\ \mapsto\ e'' : \kappa''}$$

Figure 9.4: Flow analysis with sums

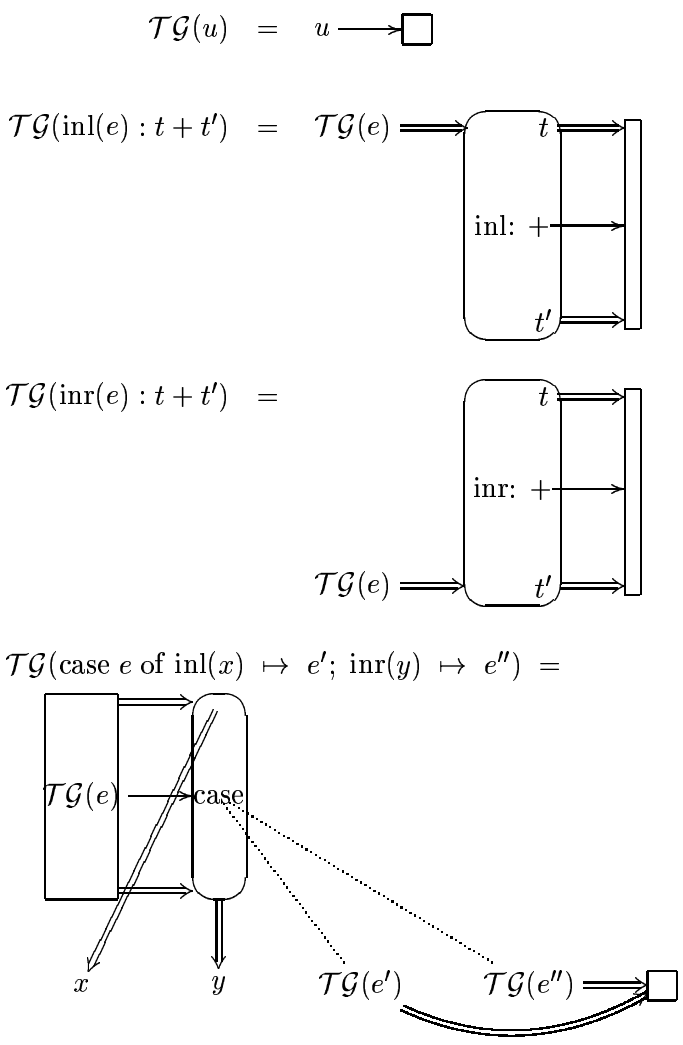


Figure 9.5: Typed flow graphs for sum types

In figure 9.5 we extend the definition of $\mathcal{TG}$ with the new syntactic constructs[1]. The constructs should be straightforward: the unit u is treated like other constants (True and False), $\text{inl}(e)$ and $\text{inr}(e)$ simply connect the root of $\mathcal{TG}(e)$ to the appropriate sub-cable of the sum-cable — note that nothing flows into the other sub-cable. Finally, the case-construct decomposes the sum (in a manner similar to $\times^-$) and connects the branches to the root (similarly to Bool^-).

9.3 Recursive Types

Recursive types add the ability to define integers, lists etc. Standard types are

$$t \quad ::= \quad \tau \mid \text{Bool} \mid t \to t' \mid t \times t' \mid t + t' \mid \mu\tau.t$$

where we have retained the sum types from above (since they are required to make practical use of recursive types). We use τ for type variables.

Usually, recursive types are added to type systems by adding the equivalence:

$$\mu\tau.t = t[\mu\tau.t/\tau]$$

Since we are decorating standard type derivations, it is convenient to make applications of this equivalence explicit in the language by adding

$$e \quad ::= \quad \text{fold}(e) \mid \text{unfold}(e)$$

with the following (standard) type rules:

$$\text{fold}\ \frac{A \vdash e : t[\mu\tau.t/\tau]}{A \vdash \text{fold}(e) : \mu\tau.t} \qquad \text{unfold}\ \frac{A \vdash e : \mu\tau.t}{A \vdash \text{unfold}(e) : t[\mu\tau.t/\tau]}$$

Example 9.1 Lists with elements of type t have the type $\mu\tau.((t \times \tau) + 1)$. The term $\text{fold}(\text{inr}(u))$ is the empty list:

$$\dfrac{\dfrac{\dfrac{}{\vdash u : 1}}{\vdash \text{inr}(u) : t \times (\mu\tau.t \times \tau + 1) + 1}}{\vdash \text{fold}(\text{inr}(u)) : \mu\tau.t \times \tau + 1}$$

The list $[\text{True}, \text{False}] : [\text{Bool}]$ is formally

$$\text{fold}(\text{inl}((\text{True}, \text{fold}(\text{inl}((\text{False}, \text{fold}(\text{inr}(u))))))))) : \mu\tau.((\text{Bool} \times \tau) + 1)$$

□

[1] Using $+^+$ and $+^-$ would have been more consistent with the rest of the naming, but would give us two $+^+$ constructs (not mentioning the rather dubious name itself).

By adding annotations to 'fold' and 'unfold' we can trace the flow of foldings to unfoldings — we do, however, not see any usefulness of such information so we leave fold and unfold unannotated. Thus we add the following rules to our formulae (see figure 9.4)

$$\tau\ \frac{}{\tau \in \mathcal{K}^{\leq}(\tau)} \qquad \mu\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \tau \in \mathcal{K}^{\leq}(\tau)}{\mu\tau.\kappa \in \mathcal{K}^{\leq}(\mu\tau.t)}$$

The inference rules for flow analysis are straightforward since no annotations are involved:

$$\text{fold}\ \frac{C; A \vdash^{\leq} e : \kappa[\mu\tau.\kappa/\tau]}{C; A \vdash^{\leq} \text{fold}(e) : \mu\tau.\kappa} \qquad \text{unfold}\ \frac{C; A \vdash^{\leq} e : \mu\tau.\kappa}{C; A \vdash^{\leq} \text{unfold}(e) : \kappa[\mu\tau.\kappa/\tau]}$$

Note, how applications of the fold rule lose information by requiring that all occurrences of κ in $\kappa[\mu\tau.\kappa/\tau]$ are annotated in the same way.

To illustrate this consider the list example above. The list $[\text{True}^{l_1}, \text{False}^{l_2}]$ could be shorthand for

$$\text{fold}(\text{inl}^{l_3}((\text{True}^{l_1}, \text{fold}(\text{inl}^{l_4}((\text{False}^{l_2}, \text{fold}(\text{inr}^{l_5}(u^{l_0})))^{l_6})))^{l_7}))$$

which we can give type

$$\mu\tau.\text{Bool}^{\{l_1,l_2\}} \times^{\{l_6,l_7\}} \tau +^{\{l_3,l_4,l_5\}} 1^{\{l_0\}}$$

Note how information is lost: we can only give a description of the elements of a list that fits all elements.

Lists have different representations. Alternatives to the type given for lists in example 9.1 include:

$$t \times (\mu\tau.t \times \tau + 1) + 1$$

and

$$\mu\tau.t \times (t \times \tau) + t \times 1 + 1$$

These types are equivalent to the type given above, but if one of these is the type used for lists in a standard typed program, it will allow greater precision at the flow analysis level. The first type will allow us to annotate the first element of lists differently from the remaining (which all need to have the same annotation). The second type requires every second element of lists to have the same annotations (that is, the first, third, fifth etc. have the same annotation and the second, fourth, sixth etc. have the same annotation).

9.3.1 Recursive Types in Typed Graphs

The same idea for treating standard type variable employed with polymorphic types is applicable to recursive types. We will adapt the variant where

τ-cables are empty — this makes even more sense with recursive types as no variable will ever have type τ and hence the values that a variable can evaluate to can be read from the graph even without τ-cables.

As with polymorphic types, the important connections are made *inside* the nodes involving τ-cables: with 'fold' and 'unfold'. The connections made are somewhat more complicated as the τ binder appears on both "sides" of the node.

A $\mu\tau.t$ cable has to carry the information from all unfoldings of the type, hence we need a t cable to carry the information of t as well as instantiations with $\mu\tau.t$ of positive occurrences of τ, and a flipped t cable to carry the information of instantiations of negative occurrences of τ. Similarly, we need a wire in each direction carrying 'fold' values. Thus

A $\mu\tau.t$ cable is  where  is a t-cable, is its flipped version and is a t-cable.

The forward single edge is the carrier and carries the label of the applied fold operation; the backward single edge carries the labels of all fold operation that can occur in argument position.

Fold and unfold m-nodes are dual and parameterised by the recursive type involved. An unfold m-node has an incoming $\mu\tau.t$ cable and an outgoing $t[\mu\tau.t/\tau]$ cable. Let superscripts index the occurrences of τ in t as in the polymorphic case. We connect the edges of the positive sub cable of the incoming cable to the nodes of the outermost t on the outgoing side. Furthermore, the incoming $\mu\tau.t$ cable is connected to all $\mu\tau.t^{(i)}$ — directly if $\tau^{(i)}$ is a positive occurrence of τ in t and "switched" if $\tau^{(i)}$ is a negative occurrence. The fold m-node has an incoming $t[\mu\tau.t/\tau]$ cable and an outgoing $\mu\tau.t$ cable. Connections are made similarly.

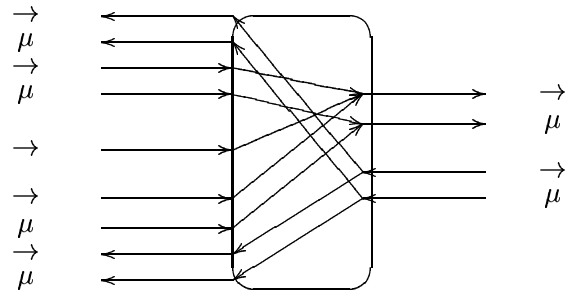
Example 9.2 The 'fold' m-node for folding $(\mu\tau.\tau \to \tau) \to (\mu\tau.\tau \to \tau)$ to $\mu\tau.\tau \to \tau$ and the dual 'unfold' m-node for unfolding $\mu\tau.\tau \to \tau$ to $(\mu\tau.\tau \to \tau) \to (\mu\tau.\tau \to \tau)$ are given in figure 9.6. The type constructors are included to remind the reader of the kind of labels carried by the individual wires. □

9.4 Dynamic Types

This thesis has so far concerned analysis of typed programs. This section will show that this need not be a restriction of the applicability of the analyses. Most programming languages that we usually consider untyped are *dynamically* typed: they perform runtime checks of well-typedness[2].

[2] An obvious exception to this is machine code.

Fold:



Unfold:

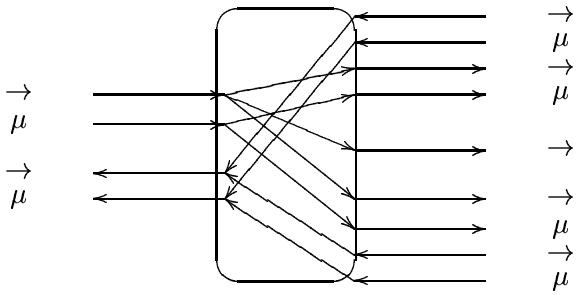


Figure 9.6: Fold and unfold

It is possible to optimize a program such that only necessary run-time checks are performed: we can statically insert the explicit run-time checks. Among the possible completions of a program we can then choose the completion with as few checks as possible. A program that is statically typable will include no explicit checks and other programs will only contain them in places where the static analysis cannot guarantee that no type errors will occur. Expressions where the analysis cannot infer a standard type will be given type *dynamic* which we write D. For more details see Henglein [Hen92, Hen94], and for a description of an extension with polymorphism, see Rehof [Reh95].

The types are thus standard types plus the special type D:

$$t \quad ::= \quad \mathsf{D} \mid \mathrm{Bool} \mid t \to t' \mid t \times t'$$

The full type system is given in figure 9.7.

Conversion Rules:

$$\begin{array}{ll} Bool! : \mathrm{Bool} \rightsquigarrow \mathsf{D} & Bool? : \mathsf{D} \rightsquigarrow \mathrm{Bool} \\ Fun! : \mathsf{D} \to \mathsf{D} \rightsquigarrow \mathsf{D} & Fun? : \mathsf{D} \rightsquigarrow \mathsf{D} \to \mathsf{D} \\ Pair! : \mathsf{D} \times \mathsf{D} \rightsquigarrow \mathsf{D} & Pair? : \mathsf{D} \rightsquigarrow \mathsf{D} \times \mathsf{D} \end{array}$$

$$\frac{c_1 : t_1 \rightsquigarrow t_1' \quad c_2 : t_2 \rightsquigarrow t_2'}{c_1 \times c_2 : t_1 \times t_2 \rightsquigarrow t_1' \times t_2'} \qquad \frac{c_1 : t_1 \rightsquigarrow t_1' \quad c_2 : t_2 \rightsquigarrow t_2'}{c_1 \to c_2 : t_1' \to t_2 \rightsquigarrow t_1 \to t_2'}$$

Non-logical rules:

$$\mathrm{Sub}\ \frac{A \vdash e : t \quad c : t \rightsquigarrow t'}{A \vdash e : t'}$$

The remaining rule are as in figure 1.1

Figure 9.7: Dynamic Typing

The conversions $Bool!, Fun!, Pair!$ correspond to tagging operations: they take an untagged value which the type system guarantees has a given type (eg. Bool) and throws it into the common pool of values about which the type system knows nothing. In this pool values have tags that can be checked at run time. Conversions $Bool?, Fun?, Pair?$ check the tag of a value and provide the untagged value of which the type inference now knows the type.

Using recursive types and sum types, we already have sufficient power to specify dynamic types. Type D is equivalent to

$$\mu\tau.(\tau \to \tau) + ((\tau \times \tau) + \mathrm{Bool})$$

The conversions are expressible in our language. E.g. *Bool*! is

$$\text{fold} \circ \text{inr} \circ \text{inr}$$

and *Bool*? is

$$\text{outr} \circ \text{outr} \circ \text{unfold}$$

where outr is a shorthand for

$$\begin{array}{l}\lambda x.\ \text{case } x \text{ of} \\ \quad \text{inl}(y) \ \mapsto\ \mathit{error}; \\ \quad \text{inr}(z) \ \mapsto\ z\end{array}$$

Using this encoding, we are able to derive a "canonical" annotation of conversions. Type D should be given *five* annotations according to the recursive type it represents:

$$\mu\tau.(\tau \to^{\ell_1} \tau) +^{\ell_4} ((\tau \times^{\ell_2} \tau) +^{\ell_5} \text{Bool}^{\ell_3})$$

If an expression e has this type, the annotation ℓ_1 represents the set of function values that e can evaluate to. Similarly, ℓ_2 and ℓ_3 represent the set of pairs respectively the set of booleans that e can evaluate to. Labels ℓ_4 and ℓ_5 give a unique identification of the tagging operations that have lead the the dynamic type. Thus these would allow us to trace the flow of tagging operations to untagging operations. For this purpose, it would be more convenient to use just a single annotation on the tagging operation. In the analysis we present in figure 9.8 we choose to leave out the tagging labels as it was not our original intention to trace this kind of flow.

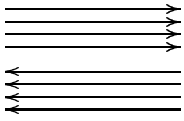
Note that we used the coding of dynamic types using recursive and sum types as guidance in the specification, but, naturally, we do not need to include these in the system.

This analysis gives as good results as subtyping flow analysis for programs that can be typed using a standard type system (i.e. where type D need not be used), but we lose information whenever a standard type cannot be inferred.

9.4.1 Dynamic Types in Typed Graphs

By the coding of dynamic types using sums and recursive types, we find that a D-cable consists of 6 forward and 6 backward edges (the 6 are $\to$, $+$, $\times$, $+$, Bool, μ).

Since the labels carried by the sum and μ edges are the same, we can replace them with one forward and one backward edge carrying tagging labels[3]:



[3]We could, as above, chose to leave tagging labels out completely.

Formulae:

$$\text{Bool}\ \frac{}{\text{Bool}^{\ell} \in \mathcal{K}^{\leq}(\text{Bool})}$$

$$\to\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \kappa' \in \mathcal{K}^{\leq}(t')}{\kappa \to^{\ell} \kappa' \in \mathcal{K}^{\leq}(t \to t')} \qquad \times\ \frac{\kappa \in \mathcal{K}^{\leq}(t) \quad \kappa' \in \mathcal{K}^{\leq}(t')}{\kappa \times^{\ell} \kappa' \in \mathcal{K}^{\leq}(t \times t')}$$

$$\text{Dyn}\ \frac{}{\mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \in \mathcal{K}^{\leq}(\mathsf{D})}$$

Conversion Rules:

$$\begin{array}{l} \mathit{Bool}! : \text{Bool}^{\ell} \rightsquigarrow \mathsf{D}^{(\{\},\{\},\ell)} \\ \mathit{Bool}? : \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \rightsquigarrow \text{Bool}^{\ell_3} \end{array}$$

$$\begin{array}{l} \mathit{Fun}! : \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \to^{\ell_1} \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \rightsquigarrow \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \\ \mathit{Fun}? : \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \rightsquigarrow \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \to^{\ell_1} \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \end{array}$$

$$\begin{array}{l} \mathit{Pair}! : \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \times^{\ell_2} \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \rightsquigarrow \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \\ \mathit{Pair}? : \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \rightsquigarrow \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \times^{\ell_2} \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \end{array}$$

$$\frac{c_1 : \kappa_1 \rightsquigarrow \kappa_1' \quad c_2 : \kappa_2 \rightsquigarrow \kappa_2'}{c_1 \times c_2 : \kappa_1 \times \kappa_2 \rightsquigarrow \kappa_1' \times \kappa_2'} \qquad \frac{c_1 : \kappa_1 \rightsquigarrow \kappa_1' \quad c_2 : \kappa_2 \rightsquigarrow \kappa_2'}{c_1 \to c_2 : \kappa_1' \to \kappa_2 \rightsquigarrow \kappa_1 \to \kappa_2'}$$

Type-specific logical rules:

$$\text{Dyn}\ \frac{C \vdash \ell_1 \subseteq \ell_1' \quad C \vdash \ell_2 \subseteq \ell_2' \quad C \vdash \ell_3 \subseteq \ell_3'}{C \vdash^{\leq} \mathsf{D}^{(\ell_1,\ell_2,\ell_3)} \leq \mathsf{D}^{(\ell_1',\ell_2',\ell_3')}}$$

Non-logical rules:

$$\text{Sub}\ \frac{C; A \vdash^{\leq} e : \kappa \quad c : \kappa \rightsquigarrow \kappa'}{C; A \vdash^{\leq} e : \kappa'}$$

Figure 9.8: Flow analysis and dynamic typing

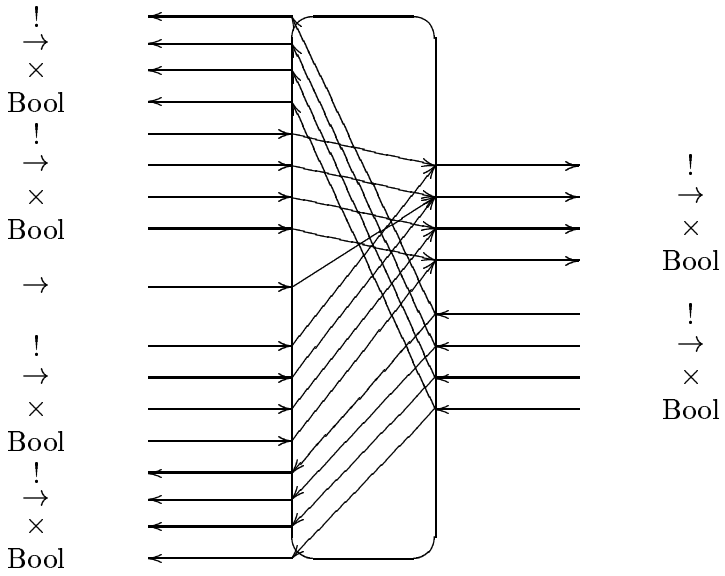


Figure 9.9: The $[Fun!]^l$ m-node

where the edges read from top to bottom carry labels of type: !, $\rightarrow$, $\times$, Bool, !, $\rightarrow$, $\times$ and Bool. Now the tagging and untagging m-nodes can be found by combining m-nodes for sum and recursive types.

Example 9.3 The $[Fun!]^l$ m-node (equivalent to $\text{fold}^l \circ \text{inl}^l$) is given in figure 9.9 (where the left-hand side is a $\mathsf{D} \rightarrow \mathsf{D}$ cable and the right-hand side is a D cable).

□

Chapter 10

Applications

This chapter will show how flow analysis can be used as the basis for program analysis. We will also illustrate the limitations of our approach imposed by our choice of predicting flow under any reduction strategy.

First we describe *constant propagation* in section 10.1. This is a classical optimisation that allows variables that can only result in *one* value to be replaced by this value. We can generalise this to replace any *expression* that can only result in a single value by the value. We will give the general construction and show how this can be used on top of the various flow analyses that we have described in this thesis.

Section 10.2 describes a generalisation of constant propagation called *firstification*. Firstification transforms a higher-order program into a first-order program by propagating all functions (along with the bindings for free variables).

In section 10.3 we will show how *binding-time* analysis can be based on our flow analyses. We show how certain properties of flow analysis are transferred to the binding-time analysis and how this imposes extra requirements on the specialiser employed.

10.1 Constant Propagation

Constant propagation analysis seeks to find expressions that will evaluate to the same value during any evaluation of the program. If this is the case, the expression can be replaced by the value. An expression can be replaced by a value v if the flow type of the expression identifies a unique value.

- Type Bool^L identifies the unique value v if $\{v\} = \bigcup_{l \in L} \text{ExpOf}(l)$.
- Type $\kappa \times^\ell \kappa'$ identifies the unique value (v, v') if κ identifies the unique value v and κ' identifies the unique value v'.
- Type $\kappa' \to^\ell \kappa$ identifies the unique value $\lambda x.v$ if κ identifies the unique value v.

It does not seem difficult to prove constant propagation based on flow analysis sound under any reduction order using the subject reduction property for the flow analysis. Some care must be taken concerning free variables (see section 10.1.1 below).

We go on to sketch how the analyses presented in chapters 2 to 6 can be used as a basis for constant propagation.

10.1.1 Simple and Subtype Constant Propagation

The above definition gives a complete definition of a constant propagation analysis based on simple or subtype flow analysis. We notice that choosing the minimal, principal flow derivation is important to obtain good results:

1. If we choose the minimal (non-principal) type derivation, the result can be unsound. Consider the minimal type judgement

   $$;\vdash^{\leq} \lambda y.\text{let } f = \lambda x.x \text{ in } (f@\text{True}^{l_1}, f@y) : \text{Bool}^{\{\}} \to (\text{Bool}^{\{l_1\}} \times \text{Bool}^{\{\}})$$

   We find that x has type $\text{Bool}^{\{l_1\}}$ but in general it would not be sound to replace x with True^{l_1}. It can be argued that it is sound under the assumption that the expression is never applied, but this is not a very useful result of constant propagation: return an optimised program that only works right if never used!

2. If the derivation is principal but not minimal, the result will naturally be less constant propagation.

We can extend the notion of constant propagation to include propagation of free variables. In minimal flow derivation, we have by theorem 2.8 that the final judgement $C; A \vdash^s e : \kappa$ contains only flow variables occurring in A or κ. Suppose that the assumption for a free variable x is $x : \text{Bool}^{\alpha}$ and α does not occur anywhere else in A or κ. Then, if an expression has type Bool^{α} in the derivation, we can replace this expression by x (eg. optimise 'if x then x else x' to 'x').

10.1.2 Polymorphic Constant Propagation

Constant propagation based on polymorphic flow analysis comes in two flavours: *sticky* or *non-sticky*. The choice depends on how the compiler can make use of the inferred information.

Sticky polymorphic constant propagation uses the idea of section 5.2.2 to obtain a ground flow function from a polymorphic derivation: replace any bound label variable α with the union of the sets of labels to which it can be instantiated.

When all polymorphism has been removed, monovariant constant propagation as described in section 10.1.1 can be applied.

Example 10.1 Consider

$$\begin{array}{l}\text{let } id = \lambda^{l_3} x.x \\ \text{in } \text{ let } y = id@\text{True}^{l_1} \\ \quad\quad \text{in } (y, id@\text{False}^{l_2})\end{array}$$

With explicit flow abstraction $\Lambda\alpha.\cdots$ and instantiation, the definition of id is $\Lambda\alpha.\lambda^{l_3}x.x : \text{Bool}^{\alpha} \rightarrow^{\{l_3\}} \text{Bool}^{\alpha}$. If we apply the above procedure, we find that id has a ground type $\text{Bool}^{\{l_1,l_2\}} \rightarrow^{\{l_3\}} \text{Bool}^{\{l_1,l_2\}}$. From this type, we see that we cannot replace any value for x. The type for y, however, is not affected by this, and remains $\text{Bool}^{\{l_1\}}$ allowing us to replace True^{l_1} for the occurrence of y. Similarly we can replace the expression $id@\text{False}^{l_2}$ with False^{l_2} and the expression $id@\text{True}^{l_1}$ with True^{l_1}.

□

The non-sticky approach inserts explicit abstractions and instantiations as above, but *retains* these as part of the result of the analysis. The compiler interprets flow abstractions and instantiations as any other abstraction and application: it should, however, choose to beta-reduce all at compile time. This will result in a new copy of the body of a flow abstraction for any set of labels to which it is applied (use *memorisation* to avoid unnecessary copies). Each copy will have different constant propagation properties.

Example 10.2 Recall the expression of example 10.1. With explicit abstraction and instantiation we get

$$\begin{array}{l}\text{let } id = \Lambda\alpha\lambda^{l_3}x.x : \text{Bool}^{\alpha} \rightarrow^{\{l_3\}} \text{Bool}^{\alpha} \\ \text{in } \text{ let } y = id\{\{l_1\}\}@\text{True}^{l_1} \\ \quad\quad \text{in } (y, id\{\{l_2\}\}@\text{False}^{l_2})\end{array}$$

By beta-reducing the instantiations we get using memorisation:

$$\begin{array}{l}\text{let } id_1 = \lambda^{l_3}x_1.x_1 : \text{Bool}^{\{l_1\}} \rightarrow^{\{l_3\}} \text{Bool}^{\{l_1\}} \\ \text{in } \text{ let } id_2 = \lambda^{l_3}x_2.x_2 : \text{Bool}^{\{l_2\}} \rightarrow^{\{l_3\}} \text{Bool}^{\{l_2\}} \\ \quad\quad \text{in } \text{ let } y = id_1@\text{True}^{l_1} \\ \quad\quad\quad\quad \text{in } (y, id_2@\text{False}^{l_2})\end{array}$$

With this expression we can replace the occurrence of x_1 with True^{l_1} and the occurrence of x_2 with False^{l_2}. In this example, the additional information inferred is useless as the information inferred in example 10.1 replaces all calls to the identity function with the result, but in general, better results can be obtained at the cost of duplicating expressions.

□

Soundness of non-sticky constant propagation does not follow immediately from the subject reduction results of chapter 5. It should be straightforward to extend the reduction system with explicit flow abstractions and

instantiations and prove a subject reduction result for the explicit polymorphic analysis. This would prove the essence of correctness of the above constant propagation; we would furthermore have to prove the memorisation employed correct. Finally, the implementation should avoid copying expressions when it can lead to duplication of evaluation.

10.1.3 Intersection Constant Propagation

As with polymorphic constant propagation, we have sticky and non-sticky versions of intersection based constant propagation. In the sticky variant, an expression with intersection flow type $\bigwedge_i \mathrm{Bool}^{L_i}$ can be replaced with $\mathrm{ExpOf}(l)$ if $\bigcup_i L_i = \{l\}$.

In the non-sticky variant, an expression of type $\bigwedge_{i\in\{1,\cdots,n\}} \kappa_i$ should be treated as an n-tuple of expressions and the subtype step corresponding to $\wedge$-elimination corresponds to projections of the tuple.

10.1.4 Graph Based Constant Propagation

If we base the analysis on the typed flow graphs of chapter 4 we examine any expression to see which constants can reach it. We should also find the free input variables that can reach the expression. The *interface* of the graph can be inferred directly from the graph: it is the collection of cables from free variables and the result cable.

By the definition of constant propagation above, the question whether an expression can be replaced by a value depends on the flow type of the expression. This can be derived from the result cable c of the expression by for each edge e in c finding the set of constructors such that there is a path leaving the carrier edge of the constructor and traversing e. Note that we do not have to construct the whole flow type: we are only interested in forward, boolean edges of rank 1 (using the standard definition of rank). To find the dependence on the input, we search for paths starting in the interface and traversing e.

We find the set of constants and input variables that traverse an edge by doing a single sink transitive closure from the edge (backwards reachability).

We can take advantage of the ability of typed graphs to do query-based flow analysis in linear time (assuming constant bounded types) to check only whether certain expressions can be replaced by a constant.

10.2 Firstification

Firstification can be seen as a generalisation of constant propagation that propagates *all* functions to applications. Since the function part of an application will not always be a unique function, it is replaced by a dispatch

for choosing the appropriate function. Thus firstification transforms higher-order programs to first-order programs. For this to make sense, we must assume that all input to the program is first-order.

Since our goal is not a description of firstification techniques, but an illustration of the usefulness of flow analysis to improve the results of firstification, we will assume that we are given a lambda-lifted program. Descriptions of lambda-lifting can be found in the literature, e.g. [PJ87]. We assume that the program has the following form:

$$\begin{array}{rcl} \text{let } f_1 & = & \lambda(x_{11}, \cdots, x_{1m_1}).e_1 \\ & \vdots & \\ f_n & = & \lambda(x_{n1}, \cdots, x_{nm_n}).e_n \\ \text{in } e & & \end{array}$$

where we assume $e, e_1, \cdots, e_n$ to be lambda free. We have furthermore allowed tupling of parameters; applications are tupled similarly thus disallowing partial applications.[1]

Now firstification proceeds as follows:

1. Replace all occurrences of variables f_i in $e, e_1, \cdots, e_n$ with the constant l_i (formally some coding of the integer i, e.g. an i-tuple of True).

2. Apply $F[\![\cdot]\!]$ to $e, e_1, \cdots, e_n$ where $F[\![\cdot]\!]$ is defined as follows:

$$\begin{array}{rcl} F[\![x]\!] & = & x \\ F[\![e@(e_1, \cdots, e_m)]\!] & = & \text{case } e \text{ of} \\ & & \quad l_1 \mapsto f_1@(F[\![e_1]\!], \cdots, F[\![e_m]\!]) \\ & & \quad \vdots \\ & & \quad l_n \mapsto f_n@(F[\![e_1]\!], \cdots, F[\![e_m]\!]) \\ F[\![\text{True}^l]\!] & = & \text{True}^l \\ F[\![\text{False}^l]\!] & = & \text{False}^l \\ F[\![\text{if}^l \ e \text{ then } e' \text{ else } e'']\!] & = & \text{if}^l \ F[\![e]\!] \text{ then } F[\![e']\!] \text{ else } F[\![e'']\!] \\ F[\![(e, e')^l]\!] & = & (F[\![e]\!], F[\![e']\!])^l \\ F[\![\text{let}^l \ (x, y) \text{ be } e \text{ in } e']\!] & = & \text{let}^l \ (x, y) \text{ be } F[\![e]\!] \text{ in } F[\![e']\!] \\ F[\![\text{fix}^l x.e]\!] & = & \text{fix}^l x.F[\![e]\!] \\ F[\![\text{let}^l \ x = e \text{ in } e']\!] & = & \text{let}^l \ x = F[\![e]\!] \text{ in } F[\![e']\!] \end{array}$$

(where we can think of 'case' as syntactic sugar for a series of 'if').

We have thus arrived at a first-order program but have paid a heavy penalty in the form of a large dispatch at every application. Flow analysis

[1] For firstification as presented here to work, flow analysis should be applied to this program. The ideas, however, extend to less crude methods of firstification without the lambda-lifting transformation — in this case the result of analysing the original program will suffice.

can reduce the cost greatly: in any case statement case e of $l_1 \ldots$ where e has type κ we can reduce the dispatch to only test for labels in $ann(\kappa)$. If $ann(\kappa)$ is a singleton set, the dispatch can be eliminated.

By the assumption on input to the program being first-order, we can (potentially) reduce all dispatches. This does not mean that we have to give up modularity entirely: only if we have case e of $l_1 \ldots$ where e has type κ and $ann(\kappa) = \alpha$ we will suspend the reduction until α is instantiated.

The sticky interpretation of polymorphic flow analysis seems to make more sense than an unsticky interpretation for firstification. Basing the optimisation on typed graphs is similar: we find the set of functions that can be applied at a given application by doing backwards reachability from the top node in the application multi-node.

We end this section by noting that the first uses of flow analysis were essentially to transform programs to first-order form allowing the application of first-order analyses. Usually the transformation was implicit, that is, the program was not transformed, but the analysis relied on closure information at every application point. The binding-time analysis of early versions of the partial evaluator Similix was based on a version of the closure analysis by Sestoft [Bon91]. The value flow information derived by the closure analysis did not immediately give the binding-time results, but allowed a binding-time analysis that was essentially first-order to be used.

10.3 Binding-Time Analysis

Partial evaluation aims at reducing a program as much as possible given part of the input to the program (for a standard reference on partial evaluation, see [GJS93]). To allow self-application, partial evaluation is often based on *binding-time* analysis, which given information on the availability of input, separates the analysed program into a *static* and a *dynamic* part. The static part can be evaluated at *specialisation* time whereas the dynamic part has to be *residualized* (i.e. made part of the specialised program). The input to binding time analysis is a program and a separation of the input into a static and a dynamic part. A subexpression of the analysed program should be annotated as static, if it can be reduced on the basis of the static input only.

10.3.1 Type-Based BTA

Let us assume that a principal flow derivation with final judgement $C; A \vdash e : \kappa$ is given where the annotations in A and the annotations occurring negatively in κ are unique label variables LV.[2] Let a binding-time separation

[2] This is the interesting case, but it is easy to extend the ideas to any proper environment.

be a division of LV into a static part LV_s and a dynamic part LV_d.

We interpret annotations as follows:

$$\begin{array}{lcll} BT(\ell) & = & \mathsf{D} & \text{, if } \exists \alpha \in LV_d \text{ s.t. } C \vdash \alpha \subseteq \ell \\ BT(\ell) & = & \mathsf{S} & \text{, otherwise} \end{array}$$

We find the binding-time of a constructor $e : \kappa$ by interpreting $ann(\kappa)$. The binding-time of a destructor e is the interpretation of $ann(\kappa)$ where $e' : \kappa$ is the destructed expression. E.g. the annotation of an application $e@e'$ is $BT(\ell)$ if e has type $\kappa \to^{\ell} \kappa'$. Other expressions ('let' and 'fix') can always be annotated as static[3]

This approach extends directly to the non-sticky polymorphic analysis: the abstractions and bound label variables are left in the program as explicit abstractions and applications. Note that we can prove BT of an annotation to be dynamic even if it involves a bound label variable. E.g.

$$\begin{array}{l} \text{let } f = \lambda^{l_\lambda} x.\text{if } d_1 \text{ then } x \text{ else } d_2 \\ \text{in } \cdots \end{array}$$

where d_1 and d_2 are free dynamic variables. The flow type of f could be

$$\forall(\alpha, \alpha_{res}).\{\alpha \subseteq \alpha_{res}, \alpha_{d_2} \subseteq \alpha_{res}\} \Rightarrow \text{Bool}^{\alpha} \to^{\{l_\lambda\}} \text{Bool}^{\alpha_{res}}$$

but we can prove that α_{res} must be D and thus the binding time type can be $\forall\alpha.\{\alpha \subseteq \mathsf{D}, \alpha_{d_2} \subseteq \mathsf{D}\} \Rightarrow \text{Bool}^{\alpha} \to^{\mathsf{S}} \text{Bool}^{\mathsf{D}}$ which in turn can be reduced to $\forall\alpha.\text{Bool}^{\alpha} \to^{\mathsf{S}} \text{Bool}^{\mathsf{D}}$. We leave the details to future work.

A kind of soundness follows from the soundness of flow analysis: if e has type κ, $\ell = ann(\kappa)$ and $\nexists\alpha \in LV_d$ s.t. $C \vdash \alpha \subseteq \ell$ then e can only evaluate to statically known values. This soundness concept of binding-time analysis is independent of the specialiser that will make use of the result of the analysis and therefore is no guarantee that a specialiser can actually make use of the information.

For this reason, binding-time analysis is often proven correct w.r.t. a given specialiser. This approach is called monolithic safety. In contrast to this Henglein and Sands discuss model based safety criteria [HS95]. Unfortunately, their criterion is not strong enough to prove even the simplest of the binding-time analyses based on flow analysis we present below (this is due to their inability to account for context-propagating specialisers, see below). Instead of giving any formal soundness results for our binding-time analyses, we will discuss informally which requirements the analysis makes on the specialiser, if a monolithic soundness proof is to be possible.

The problem with soundness can be paraphrased as *"the more precise the binding-time analysis, the harder the specialiser has to work"*. This is

[3]This is a sound annotation, though not always desirable, as it might lead to unwanted or even infinite duplication. We believe that such concerns should be left to a separate termination analysis.

very different from the constant propagation transformation, where a more precise analysis was only an asset that gave rise to better transformations. In binding-time analysis a more precise analysis might even be undesirable if no specialiser taking advantage of the information can be found.

We illustrate this by an example:

$$\begin{array}{l}\text{let } app = \lambda^{l_1} f.\lambda^{l_2} x.f@^{l_3} x \\ \text{ in if}^{l_4}\ app@^{l_5}(\lambda^{l_6} y.y)@^{l_7}\text{True}^{l_8} \text{ then } app@^{l_9} g@^{l_{10}}\text{False}^{l_{11}} \text{ else } \cdots\end{array}$$

where g is free dynamic function (g has type $\text{Bool}^{\alpha_1} \rightarrow^{\alpha_2} \text{Bool}^{\alpha_3}$ where $\alpha_1, \alpha_2, \alpha_3 \in LV_d$). A simple flow analysis as well as a subtype flow analysis will infer that both applications of app will result in some α where $\alpha_3 \subseteq \alpha$ and $\{l_8\} \subseteq \alpha$. Hence the conditional and the branch need to be made dynamic.

The annotated program resulting from subtype based flow analysis will be:

$$\begin{array}{l}\text{let } app = \lambda^{\mathsf{S}} f.\lambda^{\mathsf{S}} x.f@^{\mathsf{D}} x \\ \text{ in if}^{\mathsf{D}}\ app@^{\mathsf{S}}(\lambda^{\mathsf{S}} y.y)@^{\mathsf{S}}\text{True}^{\mathsf{S}} \text{ then } app@^{\mathsf{S}} g@^{\mathsf{S}}\text{False}^{\mathsf{S}} \text{ else } \cdots\end{array}$$

The simple flow analysis based binding-time analysis will require True and False as well as λ^{l_6} to be dynamic:

$$\begin{array}{l}\text{let } app = \lambda^{\mathsf{S}} f.\lambda^{\mathsf{S}} x.f@^{\mathsf{D}} x \\ \text{ in if}^{\mathsf{D}}\ app@^{\mathsf{S}}(\lambda^{\mathsf{D}} y.y)@^{\mathsf{S}}\text{True}^{\mathsf{D}} \text{ then } app@^{\mathsf{S}} g@^{\mathsf{S}}\text{False}^{\mathsf{D}} \text{ else } \cdots\end{array}$$

The binding-time result from simple flow analysis can be interpreted by standard specialisers (and we believe this to be true for all programs), but even the sub-type based result requires non-standard techniques. Consider the variables x and f. The flow type of x is $\text{Bool}^{\{l_8,l_{11}\}}$ and thus x is static. The flow type of f is $\text{Bool}^{\{l_8,l_{11}\}} \rightarrow^{\alpha_f} \text{Bool}^{\alpha_{f_{res}}}$ where $\alpha_2 \subseteq \alpha_f$, $\{l_6\} \subseteq \alpha_f$, $\alpha_3 \subseteq \alpha_{f_{res}}$ and $\{l_8, l_{11}\} \subseteq \alpha_{f_{res}}$. This implies that the binding time type of f is $\text{Bool}^{\mathsf{S}} \rightarrow^{\mathsf{D}} \text{Bool}^{\mathsf{D}}$. The flow type of $\lambda^{l_6} y.y$ is $\text{Bool}^{\{l_8,l_{11}\}} \rightarrow^{\{l_6\}} \text{Bool}^{\{l_8,l_{11}\}}$ which maps to the binding-time type $\text{Bool}^{\mathsf{S}} \rightarrow^{\mathsf{S}} \text{Bool}^{\mathsf{S}}$.

The problem is that while the subtyping step

$$\text{Bool}^{\{l_8,l_{11}\}} \rightarrow^{\{l_6\}} \text{Bool}^{\{l_8,l_{11}\}} \leq \text{Bool}^{\{l_8,l_{11}\}} \rightarrow^{\alpha_f} \text{Bool}^{\alpha_{f_{res}}}$$

is legal (under the assumption given above), the corresponding binding-time subtype step

$$\text{Bool}^{\mathsf{S}} \rightarrow^{\mathsf{S}} \text{Bool}^{\mathsf{S}} \leq \text{Bool}^{\mathsf{S}} \rightarrow^{\mathsf{D}} \text{Bool}^{\mathsf{D}}$$

needs to be interpreted by the specialiser. The point is that subtype steps have an operational meaning in partial evaluation, namely converting the internal representation of a value to the program fragment representing the

value. While it is straightforward to convert the internal representation of a first-order value to its program representation, it is not as easy to convert the internal representation of a function (i.e. a closure) to its external representation (i.e. a lambda-expression). Therefore, most specialisers disallow higher-order subtype steps (coercions) as the step above.

Higher-order coercions *can* be handled by interpreting them as eta-conversions[4]. Above, the binding-time analysis would transform $\lambda^{\mathsf{S}}y.y$ to $\lambda^{\mathsf{D}}z.(\lambda^{\mathsf{S}}y.y)@^{\mathsf{S}}z$. Thus, the binding-time analysis based on subtyping flow analysis remains sound, but puts constraints on the specialiser that can utilise the derived binding-time information.

As an aside, we note that with higher-order coercions, all constructors can be annotated as static. This implies that we are only concerned with finding the annotations for destructors — thus binding-time analysis can be based directly on a flow function $\mathcal{F}$.

Binding-time analysis based on non-sticky polymorphic flow analysis requires the specialiser to be able to interpret *explicit* binding-time abstraction and instantiation. The above example is annotated as follows:

$$\begin{array}{l}\text{let } app = \Lambda(\alpha_x, \alpha_f, \alpha_{f_{arg}}, \alpha_{f_{res}})\lambda^{\mathsf{S}}f.\lambda^{\mathsf{S}}x.f@^{\alpha_f}x \\ \text{ in } \ \text{if}^{\mathsf{S}}\ app\{\mathsf{S},\mathsf{S},\mathsf{S},\mathsf{S}\}@^{\mathsf{S}}(\lambda^{\mathsf{S}}y.y)@^{\mathsf{S}}\text{True}^{\mathsf{S}} \\ \qquad \text{then } app\{\mathsf{S},\mathsf{D},\mathsf{D},\mathsf{D}\}@^{\mathsf{S}}g@^{\mathsf{S}}\text{False}^{\mathsf{S}} \\ \qquad \text{else } \cdots\end{array}$$

A specialiser capable of handling explicit abstraction and instantiation is described in [HM94] — the extension is straightforward: treat binding-time abstraction and instantiation like static lambda-abstraction and application.

If the above expression is analysed using a sticky binding-time analysis based on polymorphic flow analysis, we find the following annotation

$$\begin{array}{l}\text{let } app = \lambda^{\mathsf{S}}f.\lambda^{\mathsf{S}}x.f@^{\mathsf{D}}x \\ \text{ in if}^{\mathsf{S}}\ app@^{\mathsf{S}}(\lambda^{\mathsf{S}}y.y)@^{\mathsf{S}}\text{True}^{\mathsf{S}} \text{ then } app@^{\mathsf{S}}g@^{\mathsf{S}}\text{False}^{\mathsf{S}} \text{ else } \cdots\end{array}$$

Even though this annotation is sound in the way discussed above, it would be a very bad starting point for a specialiser as the annotation D on the application does *not* indicate that this application should *always* be residualized.

Intersection based flow analysis naturally gives rise to the most demanding binding-time analysis — since specialisers are usually required never to discard computation, our completeness result on the flow analysis implies that a destructor is only deemed dynamic if the argument (function, conditional or pair) is part of the dynamic input.

The binding-time analysis should insert explicit $\wedge$-I and $\wedge$-E constructs which would be interpreted by the specialiser as pairing and projections.

[4] A different approach is to keep both representations at all times during specialisation.

Thus any expression of type $\bigwedge_{i\in\{1,\cdots,n\}} \kappa_i$ would be copied in n versions during specialisation. The complexity of such a specialiser would be as forbidding as the complexity of the intersection based analysis.

All the above binding-time analyses have an additional requirement to the specialiser. Consider 'if'

$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

If e_1 is annotated as dynamic the conditional cannot be reduced, and a "direct-style" specialiser will then require the result of the 'if' to be deemed dynamic. This is true even if the branches depend only on static input.

We can get around this problem by improving the specialiser: a *context-propagating* or *CPS-based* specialiser can "push" the context in which the conditional occurs to the branches and allow specialisation of static branches under dynamic control — the context propagation rules allowing this correspond to the rules of figure 6.8.

While the problems described first originated from better binding-time analyses requiring better specialisers, this problem is inherent in all our flow analyses — why is this so? The answer lies in our very first decision to approximate the value flow under *any* reduction order. A direct style specialiser relies on a specific reduction order (conditional before branches) and thus requires the flow analysis to model this. Surprisingly, our choice of sacrificing some precision to achieve general applicability has led us to analyses that are *not* generally applicable!

10.3.2 Graph-based BTA

It is straightforward to base binding-time analysis on typed graphs: attach the constant D to all edges in the interface that correspond to dynamic input (see section 10.1.4 on the notion of interface). Binding-time analysis proceeds by doing single-source transitive closure (or reachability) from these constants. All destructors that terminate a path starting from D should be annotated with D.

If we assume that all types are of bounded size, this results in a linear time algorithm.

Henglein describes a near linear time algorithm (in the size of the (untyped) program) for binding-time analysis of untyped programs [Hen91]. This analysis is of strength comparable to the graph-based algorithm but differs by not allowing higher-order coercions and by assuming a direct-style specialisation of 'if'. His analysis can, however, be changed to deal with this — retaining the complexity if allowing higher-order coercions would, as in our case, rely on an assumption that all expressions had bounded types.

We note that with this approach it is also easy to do "poor man's generalisation" [Hol88]. This is an optimisation, that annotates any constructor that will always eventually be coerced to dynamic as dynamic. The point is

that if an expression will always be residualized, it can be an advantage to do so as early as possible.

The binding-time analysis has divided the destructors into static and dynamic by initially assuming all to be static, and then deeming as few as possible dynamic using forward reachability from the set of dynamic inputs. We implement "poor man's generalisation" in the same way: initially assume all constructors to be dynamic and do a backward reachability from the set of static destructors thus deeming only the constructor that will actually be used statically as static. As with binding-time analysis, this can be done in linear time assuming that types are of bounded size.

Chapter 11

Related Work

In this chapter we will discuss the relation to other work. We will not discuss the vast body of work on flow analysis of first-order programs. Instead, we concentrate on previous and contemporary work on flow analysis of higher-order functional languages. We have — somewhat artificially — divided the work into abstract interpretation based (section 11.1), constraint based (section 11.2), set-based (section 11.3) and type based analyses (section 11.4). The development of flow analyses has often gone hand in hand with work on ensuring type safety of programs. Section 11.5 briefly presents and discusses work in this area.

11.1 Abstract Interpretation

The first (to our knowledge) flow analysis of higher-order programs was defined by Jones [Jon81a, Jon81b]. Jones defines an abstract machine implementing (outside-in) call-by-value reduction. The machine rewrites states consisting of a context (identifying the redex within the term) and a redex. To avoid substitution, expressions are represented by a pair of a term and an environment mapping free variables to values. Using a finite approximation of states, a flowchart can be achieved such that, if there is a state transition in the evaluation of a term, then there is an arrow in the flowchart between the abstract values of the states. This kind of flow analysis can truly be called control flow analysis, as evaluation order under call-by-value is modelled in the flow chart. Furthermore, the abstract states contain information about the values of free variables in closures.

A later flow analysis by Jones analyses lambda lifted (no explicit lambdas, but partial applications allowed) lazy functional programs [Jon87]. The analysis finds safe descriptions of input-output behaviour of each defined function: a grammar is used to give an approximate description of the values that can be bound to variable and the value that a function can result in.

The closure analysis of Sestoft [Ses88, Ses91] was developed to facilitate *globalisation analysis* [Ses89], which attempts to discover whether a function parameter can be turned into a global variable. We have described the analysis in depth in section 3.3 and will not describe it any further here. Sestoft's analysis formed the basis for the closure analysis in Similix-2 [Bon91] where the information was used as a basis for a number of analyses, most notably *binding-time analysis*.

Ayers extended the Sestoft's closure analysis with reachability as described in section 8.1 [Aye92].

Independently of Sestoft, Shivers [Shi88, Shi91c, Shi91b] defined the flow analysis described in chapter 7. We will not describe this further here, as it has already been covered in depth.

Jaganathan and Weeks describe a general framework for development of flow analyses [JW95a]. They analyse a higher-order language with constructors and first-class references, and give an exact operational semantics using a set of states and a transition function. Analyses are obtained by abstract interpretation of the semantics. An analysis equivalent to the analyses of chapter 3 (mistakenly coined 0CFA) is described as one abstraction and is proven equivalent to *set-based* analysis (see below). As in Shivers' analyses, contours are used to allocate new dynamic activation frames and they are represented by a list of call-site labels. This allows abstractions in the style of Shivers' nCFA, but the framework allows greater variation in abstraction, and a non-trivial abstraction of contours is described that does not lead to exponential complexity. Even though alternative representations of contours are discussed, it is not clear that the framework is sufficiently general to include analyses such as our polymorphic and intersection based analyses.

Jaganathan and Wright describe a flow analysis which strictly improves over Sestoft's analysis [JW96b]. The first improvement is an extension with reachability like Ayers', but more importantly they describe a technique called polymorphic splitting: they use contours like Shivers to record the call-history, but inspired by let-polymorphism, only calls to let-bound procedures are recorded. Thus the call-history is bounded by the nesting depth of 'let' and no abstraction (like in nCFA) has to be made. In its full generality, this idea corresponds to analysing a fully let-unfolded program and hence to our let-polymorphic flow analysis of section 5.2. This is, however, deemed prohibitively complex and the analysis given yields less precise information than our let-polymorphic analysis in cases such as:

$$\begin{array}{l}\text{let } f = \lambda x.x \\ \text{ in } \text{ let } g = \lambda x.\lambda y.(f@x, f@y) \\ \qquad \text{in } (g@\text{True}^{l_1}@\text{True}^{l_2}, g@\text{False}^{l_3}@\text{False}^{l_4})\end{array}$$

(adapted from an example in [JW96b]) which will merge the results of the first occurrence of f resulting from the two calls to g, and similarly the

results of the second occurrence. The flow description of the full expression will thus be $(\text{Bool}^{\{l_1,l_3\}} \times \text{Bool}^{\{l_2,l_4\}}) \times (\text{Bool}^{\{l_1,l_3\}} \times \text{Bool}^{\{l_2,l_4\}})$ instead of the precise result $(\text{Bool}^{\{l_1\}} \times \text{Bool}^{\{l_2\}}) \times (\text{Bool}^{\{l_3\}} \times \text{Bool}^{\{l_4\}})$ achievable using let-polymorphic flow analysis.

The analysis has been implemented for full Scheme and reasonable run-times are reported for the untuned implementation. The paper describes applications of the analysis to avoid run-time checks and inlining (which is also reported in [JW95b] resp. [JW96a]) and shows significant speed-ups due to the analyses.

11.2 Constraint Based Analysis

Bondorf and Jørgensen described a very simple linear time flow analysis for Similix [BJ93] — we have already described this analysis in chapter 2 so we will not describe this further here.

Palsberg [Pal94] investigated the relationship between the traditional abstract interpretation based analyses and constraint based analyses. He proves that the constraint based closure analysis he presents is equivalent to the closure analysis of Bondorf [Bon91] (which in turn was based on Sestoft's analysis [Ses88, Ses91]). This leads Palsberg to a proof of preservation of flow information under arbitrary beta-reduction (Sestoft had only been able to prove invariance under call-by-name and call-by-value).

Independently of our work, Heintze and McAllester recently described an algorithm for flow analysis which allowed single queries to be answered in linear time under the assumption that the size of types was bounded [HM97]. Not only is the result similar to the result reported in section 4.2, but the algorithm is essentially identical: reachability in a graph, where each subexpression is represented by a collection of nodes (named n, $dom(n)$, $ran(n)$ etc.). The graph is not constructed directly from the explicitly typed program but using inference rules — this avoids adding argument edges to functions that are never applied and result edges for functions whose result is never used. Hence graph construction is slightly more complicated and the resulting graphs slightly smaller, but the asymptotic complexity is identical. The difference shows up in their extension to ML polymorphic languages, where edges are created corresponding to the collection of instances of a polymorphic function. This does not give any polymorphism at annotation level as our approach does (though, when they extend the analysis to handle pairs, instantiations of a type variable to pairs and functions will be kept separate thus giving some polymorphism — this fact is not noted by the authors). The authors sketch an extension of their analysis with 'cons', but while their treatment resembles ours, it seems rather ad hoc and it is not clear how it generalises to arbitrary recursive types.

11.3 Set-Based Analysis

Set-based analysis and our flow analysis have very similar aims: given a program, set-based analysis finds a set of values for each variable and subexpression that is a superset of the values that the variable resp. subexpression can evaluate to. This differs slightly from flow analysis, since it does not infer which constructor was responsible for the value — the extension, however, is trivial.

Set constraints for higher-order functional programs were introduced by Heintze [Hei90, Hei93]. His analysis works by generating set-constraints and finding a least solution to these. Heintze gives an $\mathcal{O}(n^3)$ algorithm for constraint solving. Heintze's analysis and in particular his derivation of the analysis is rather complicated. The analysis was simplified by Flanagan and Felleisen [FF95], who used a reduction semantics with a global heap. This allows easier and less ad hoc extensions of the analysis to languages with assignment and non local control constructs.

The analyses are very similar to the flow analysis of Palsberg [Pal94]. This is particularly evident in the formulation of Flanagan and Felleisen though this is not remarked upon and Heintze remarks that his analysis "can be roughly compared to 0CFA" (though this seems to be under the prevalent misconception of 0CFA and closure analysis being equivalent)[1].

The set-based analyses of Heintze and Flanagan and Felleisen do not state any results concerning modular analysis and it is far from obvious how modularity can be achieved. Both analyses seek the *least* solution under set-inclusion which, as discussed in the introduction, is the relevant solution concerning the internal flow in a program, but which is rather useless when describing the input-output behaviour and the flow of arguments: it seems likely that the analyses can be extended such that input to the program (via free variables or arguments) could be treated separately, giving them a special value which could be traced through the program (such a special value is not essentially different from our label variables).

Heintze briefly mentions an extension of set-based analysis with *polyvariance*. Essentially, this works by duplicating functions to allow separate analysis of different calls to the function. Instead of duplicating the function itself, the constraints generated from the function can be duplicated. In its full generality this can achieve the same precision as our polymorphic flow analysis, it is, however, not immediately implementable in each full generality as there is no obvious way of controlling the amount of duplication done. Heintze uses a heuristic based on a first mono-variant analysis that approximates the set of functions that can potentially benefit from duplication. This approach is far more ad hoc than our, and can (by theorems 5.24 and 5.43) never be better than let- and fix-polymorphic flow analysis. Furthermore,

[1] As mentioned above, the connection was proven by Jaganathan and Weeks [JW95a].

no complexity is given on the polyvariant analysis — in the implementation described in [Hei90] a few empirical results are given: some show little overhead in using the polyvariant version, but in other cases the overhead is rather big and for one program no timings are given due to excessive copying.

Independently of our work, recent work by Flanagan and Felleisen concerns modular and polymorphic set-based analysis of untyped languages [FF96]. The basic idea is similar to the idea of chapter 5: flow analysis generates a set of constraints containing variables. Flanagan and Felleisen do not consider polymorphic recursion. The constraint sets are partially solved while retaining observational equivalence of constraint sets — similar to finding minimal, principal solutions.

There are important differences, however: Flanagan and Felleisen find that computing a minimal equivalent of a given constraint set is extremely expensive (PSPACE-hard) and concentrate on finding efficient algorithms for simplification that do not always lead to the minimal solution. The difference is probably due to the absence of standard types in their analysis, but a closer comparison between their analysis and our let-polymorphic analysis will have to be postponed.

11.4 Type Based Analysis

Tang and Jouvelot [TJ92] describe an *escape* analysis that uses *control-flow effects*. The analysis infers for each subexpression e an approximation of the set of abstractions applied during evaluation of e and thus the aim of the analysis differs slightly from ours. The paper introduces flow variables and thus allows modular analysis; the paper does not state any principality results, and indeed such a result cannot be proven in their system since it contains subtyping, but judgements do not include constraint sets. Though not stated, the analysis is only applicable to call-by-value languages. The language analysed is the simply typed lambda calculus without recursion, but does extend the language with side-effects. The analysis is polymorphic over flow values, but only in the sense that (side-effect free) let-expressions are unfolded prior to analysis.

The aim of a later paper by the same authors [TJ94] is to combine the power of Shivers' 1CFA with the possibility of separate analysis from type based flow analyses. The idea is to analyse each module using 1CFA but with type-like assumptions on free variables, and use a type based analysis to combine the results of modules. The language is simply typed lambda calculus. The type system corresponds to the system of chapter 3. The type environment for individual modules is used to approximate the abstract interpretation environment for the module — this allows separate analysis at the cost of precision.

Independently of Palsberg and O'Keefe (see the next section), Heintze [Hei95] studied the relationship between standard type systems and flow analysis. In addition to Palsberg and O'Keefe's result, he showed that by annotating types in a straightforward manner Amadio and Cardelli's type system can be used to perform data flow analysis (the same analysis as safety analysis was based on). He went on to prove a number of equivalences:

1. Subtyping and recursive types $\equiv$ Sestoft's analysis.
2. Subtyping $\equiv$ Sestoft's analysis where cycles in the flow graph are forbidden.
3. Recursive types $\equiv$ an equality based analysis (with cycles).
4. Simple types $\equiv$ an equality based analysis without cycles as described by Bondorf and Jørgensen [BJ93].

Independently of the present work, Banerjee recently studied rank 2 intersection based flow analysis of ML-polymorphic programs [Ban97]. Banerjee does not attach flow information to the standard types of the program in the way that we do, but exploits that every ML-typable program can be given a rank 2 intersection type. Thus, a complete type inference is performed with annotated types. This prevents an assessment of the complexity of the analysis in terms of the typed program, and it is not clear whether the practical tractability of ML typing carries over to his analysis. Modular analysis is obtained by constructively proving the existence of principal typings; it is noted that the inferred constraints can be reduced to "some normal form", but no construction is given for finding minimal, principal typings (note, that in contrast to full intersection typing, rank 2 intersection has a fairly straightforward notion of instance and hence of principality). As noted in section 8.5 we believe that rank 2 intersection is an interesting compromise between practicality and precision, and we believe that combining the ideas of Banerjee with the framework of this thesis could lead to a better understanding of problems involved.

11.5 Safety, Type Recovery and Soft Typing

The development of flow analyses has often gone hand in hand with attempts to solve typing problems for untyped programs, either by providing type safety or by eliminating unnecessary run-time type checks. One of the main motivations and applications of Shivers' analyses was *type recovery* in Scheme [Shi90, Shi91c, Shi91a].

Palsberg and Schwartzbach [PS92a, PS92b] describes an analysis called *safety analysis*. The purpose of this analysis is very similar to the *type recovery* analysis of Shivers and is based on data flow analysis (in this case

on an analysis equivalent to Sestoft's [Ses88, Ses91]). Safety analysis checks that if flow analysis predicts that a set of values can end up in function position of an application, then all values are indeed functions.

Later, Palsberg and O'Keefe [PO95] showed an exact correspondence between safety analysis and Amadio and Cardelli's type system with subtyping and recursive types [AC91].

As mentioned above, Jaganathan and Wright used their polymorphic splitting flow analysis to avoid run-time checks [JW95b] and Flanagan and Felleisen uses their set-based analysis for soft typing [FF95].

Aiken, Wimmers and Lakshman defined soft typing directly [AW93, AWL94]. They define a very precise type system (including intersection and union) and only insert dynamic type checks when the analysis is unable to guarantee type correctness. Though their analysis does not infer flow information in the style of this thesis, the type system is sufficiently fine-grained to infer the flow of constructors. This allows *conditional* types τ *if* τ' and seems very similar to the reachability optimisation described for our analyses in section 8.1. It would be interesting to use their type system as a basis for a flow analysis in the style presented here.

Chapter 12

Conclusion

This chapter summarises, concludes and discusses future work.

12.1 Summary

We have presented a number of program analyses for approximating the value flow during execution of a program. The key features of the analyses were:

- **Separate Compilation:** The type based analyses of chapters 2, 3 and 5 all possessed the *principal typing* property allowing modules to be analysed separately without loss of precision. *Minimal* principal typings completely resolve all flow results that are not dependent on the context.

  Typed flow graphs (chapter 4) could be constructed in a modular manner and by adding special values to input edges reachability results could be combined to yield modular analysis.

- **Precision:** Except for the simple flow analysis of chapter 2, all analyses are at least as precise as closure analysis and polymorphism gives added precision that we expect to be useful in practice.

- **Practicality:** If we assume that all types have bounded size, the analysis based on typed graphs (chapter 4) results in better complexity (quadratic) than was previously known (cubic). The extension with polymorphism can still be handled in polynomial time.

- **Evaluation-Order Independence:** All analyses were proven to be sound under any order of evaluation.

Furthermore, we presented an analysis based on intersection types, which, while not being practical, gives exact results.

12.2 Future Work

We have already discussed some directions for future work in chapters 8 and 9. We will summarise the most important issues raised in these chapters and suggest other interesting topics.

12.2.1 Robustness

We have shown our analyses to be sound under any order of standard reduction and thus for a large number of compiler optimisations. It would be interesting to investigate soundness under other transformations such as CPS-translation, eta-conversion etc.

12.2.2 Making the Analyses more Generally Applicable

Chapter 9 discussed various extensions of the standard type system: we showed how the analyses could be extended to accommodate languages with polymorphism, sum types, recursive types and dynamic types. We did not, however, provide any proofs of correctness of these extensions. A more careful study of these extensions would also lead to a better understanding of the relationship between type based analysis with a dynamic standard type system and analyses for untyped languages.

We have not studied how the analyses could embrace imperative features, eg. assignments, references and call-cc. While we do not expect any problems with extending the analyses, a closer study is required to make the analyses applicable to impure functional languages such as Scheme or ML.

12.2.3 Improvements of the Analyses

We suggested a number of improvements of our analyses in chapter 8. We find the idea of labelling graphs to be particularly promising: such an extension would bridge the gap between flow analysis of imperative programs and functional programs. We hope that practical analyses could be developed that are exact (under the same assumptions as intersection based analysis) for first-order programs. Furthermore, such an analysis would expose an interesting link to the theory of optimal reduction.

While we find that the theory of polymorphic flow analysis is developed in depth, we still find that the given algorithm has room for improvement: the given complexity results seem overly conservative, and we believe that better algorithms can be found.

We have discussed the pros and cons of evaluation-order dependency in flow analysis, but mainly focused on evaluation-order independent analysis. A better understanding of integrating evaluation-order dependency in our analyses would expand the field of application of the analyses.

Dansk sammenfatning

Ved oversættelse af programmer er det ofte nyttigt at have information om hvorledes værdier vil blive skabt og brugt under afvikling af programmet. Formålet med flow analyse er at forudse og beskrive den mulige strøm af værdier gennem et givet program uden kendskab til dets inddata værdier.

Denne afhandling præsenterer en række flow analyser for typede, højereordens funktionsprogrammeringssprog. Fælles for analyserne er

- **Modularitet:** Enkelte moduler i et program kan analyseres separat. Dette tillader separat oversættelse af programmet.
- **Praktisk anvendelighed:** Med undtagelse af intersection baseret analyse (se nedenfor) har alle analyser polynomisk kompleksitet. Specielt præsenteres en ny metode til beregning af *closure* analyse, der er kvadratisk (under antagelse at alle typer har begrænset størrelse) imodsætning til tidligere kendte metoder, der alle var kubiske.
- **Præcision:** Ved at udvide flow analyse med *polymorfi* og *intersection* opnås analyser, der er strengt mere præcise end closure analyse (intersection baseret analyser er strengt mere præcis end og polymorf flow analyse er usammenlignelig med nCFA). Præcisionen af polymorf analyse karakteriseres ved invarians under let- og fix-expansion, og præcisionen af intersection baseret analyse karakteriseres ved invarians under generel expansion (under et ikke-standard reduktions-system, der aldrig smider beregninger væk og evaluerer begge grene i 'if').
- **Uafhængighed af evalueringsrækkefølge:** De præsenterede analyser bevises korrekte under vilkårlig evalueringsrækkefølge.

Bibliography

[AC91] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. 18th Annual ACM Symposium on Principles of Programming Languages (POPL), Orlando, Florida*, pages 104–118. ACM Press, Jan. 1991.

[ADLR94] Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. Paths in the lambda-calculus: Three years of communication without understanding. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 426–436, Paris, France, 1994. IEEE Computer Society Press.

[AG96] Andrea Asperti and Stafano Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1996. To appear.

[AL93] Andrea Asperti and Cosimo Laneve. Paths, computations and labels in the λ-calculus. In *RTA'93*, volume 690 of *LNCS*, pages 152–167. Springer-Verlag, 1993.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In David Schmidt, editor, *1993 ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 31–41. ACM, June 1993.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proc. 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon*. ACM, ACM Press, Jan. 1994.

[Aye92] Andrew Ayers. Efficient closure analysis with reachability. In *Proc. Workshop on Static Analysis (WSA), Bordeaux, France*, pages 126–134, Sept. 1992.

[Ban97] Anindya Banerjee. A modular, polyvariant, and type-based closure analysis. In *International Conference on Functional Programming*. ACM Press, 1997.

[BCDC83] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 1983.

[BDCd95] Franco Barbenera, Mariangiola Dezani-Ciancaglini, and Ugo de'Ligouro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119:202–230, 1995.

[BHA86] Geoffrey Burn, Chris Hankin, and Samson Abramsky. Strictness analysis of higher-order functions. *Science of Computer Programming*, 7:249–278, Nov. 1986.

[BJ93] Anders Bondorf and Jesper Jørgensen. Efficient analysis for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.

[Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, December 1991. Selected papers of ESOP '90, the 3rd European Symposium on Programming.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analsis of programs by construction or approximation of fixpoints. In *Proc. ACM POPL '77*, pages 238–252, 1977.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[Dam84] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984. Technical Report CST-33-85 (1985).

[DHM95a] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In Alan Mycroft, editor, *Proc. 2nd Int'l Static Analysis Symposium (SAS), Glasgow, Scotland*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, September 1995.

[DHM95b] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-

time analysis in polynomial time. Working version of [DHM95a], 1995.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[FF95] Cormac Flanagan and Matthias Felleisen. Set based analysis for full scheme and its use in soft-typing. Technical Report TR95-253, Rice University, 1995.

[FF96] Cormac Flanagan and Matthias Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Technical Report TR96-266, Rice University, November 1996.

[Gir72] J.-Y. Girard. Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supŕieur. Doctoral thesis, 1972. University of Paris VII.

[Gir89] J.-Y. Girard. Geometry of interaction 1: Interpretation of system F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium '88*, pages 221–260. North-Holland, 1989.

[GJS93] Carsten K. Gomard, Neil D. Jones, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[GRDR88] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proc. Symp. on Logic in Computer Science*, pages 61–70. IEEE, Computer Society, Computer Society Press, June 1988.

[Gun92] Carl A. Gunter. *Semantics of Programming Langauges: Structures and Techniques.* Foundations of Computer Science. MIT Press, 1992.

[Hei90] Nevin Heintze. Set-based analysis of ML programs. In *Lisp and Functional Programming*, pages 306–317, 1990.

[Hei93] Nevin Heintze. Set constraints in program analysis (survey). In *Proceedings of the* 2^{nd} *International Workshop on Principles and Practice of Constraints Programming*, 1993. invited paper.

[Hei95] Nevin Heintze. Control-flow analysis and type systems. In Alan Mycroft, editor, *Symposium on Static Analysis (SAS'95)*, volume 983 of *LNCS*, pages 189–206, Glasgow, 1995.

[Hen88] F. Henglein. Type inference and semi-unification. In *Proc. ACM Conf. on LISP and Functional Programming (LFP), Snowbird, Utah*, pages 184–197. ACM Press, July 1988.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *FPCA*, volume 523 of *Lecture Notes in Computer Science*, pages 448–472. 5th ACM Conference, Cambridge, MA, USA, Springer-Verlag, August 1991.

[Hen92] Fritz Henglein. Dynamic typing. In B. Krieg-Brückner, editor, *Proc. European Symp. on Programming (ESOP), Rennes, France*, volume 582 of *Lecture Notes in Computer Science*, pages 233–253. Springer-Verlag, February 1992.

[Hen94] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming (SCP)*, 22(3):197–230, 1994.

[Hen96] Fritz Henglein. Syntactic properties of polymorphic subtyping. Unpublished Manuscript, 1996.

[HM94] Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Proceedings of European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, April 1994.

[HM97] Nevin Heintze and David McAllester. Linear time subtransitive control flow analysis. In *Conference on Programming Language Design and Implementation (PLDI'97)*, pages 261–272. ACM SIGPLAN, 1997.

[Hol88] Carsten Kehler Holst. Poor man's generalization. Working note, August 1988.

[HR95] Kristoffer H. Rose. *Xy-pic User's Guide*, September 1995. Version 3.2.

[HS95] Fritz Henglein and David Sands. A semantic model of binding times for safe partial evaluation. In S.D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 299–320. Springer-Verlag, 1995.

[Jen92] T. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, Imperial College, Univ. of London, November 1992. Available as DIKU Report 93/11.

[Jon81a] N. Jones. Flow analysis of lambda expressions. Technical Report DAIMI PB-128, Aarhus University, Jan. 1981.

[Jon81b] Neil Jones. Flow analysis of lambda expressions. In S. Even and Kariv O, editors, *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, volume 115 of *LNCS*, pages 114–128, Acre, Israel, 1981. Springer-Verlag.

[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 103–122. Ellis Horwood, Chichester, England, 1987.

[JW95a] Suresh Jaganathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL'95*, pages 393–407. ACM Press, 1995.

[JW95b] Suresh Jaganathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In Alan Mycroft, editor, *SAS'95*, volume 983 of *LNCS*, pages 207–224. Springer-Verlag, 1995.

[JW96a] Suresh Jaganathan and Andrew Wright. Flow directed inlining. In *Symposium on Programming Language Design and Implementation*. ACM, 1996.

[JW96b] Suresh Jaganathan and Andrew Wright. Polymorphic splitting: An effective polyvariant flow analysis. Submitted to ACM Transactions on Programming Languages and Systems (TOPLAS), 1996.

[Lam90] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. 17-th Annual ACM Symposium on Principles of Programming Languages, San Francisco*, pages 16–30. ACM Press, New York, NY, January 1990.

[Lév78] M. Lévy. *Réductions correctes et optimales dans le lambda calcul.* PhD thesis, Université Paris VII, 1978.

[Lév80] M. Lévy. Optimal reductions in the lambda-calculus. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.

[Mai92] Harry G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 2(103):387–394, September 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Mos97a] Christian Mossin. Exact flow analysis. In *Fourth International Static Analysis Symposium (SAS'97)*, 1997.

[Mos97b] Christian Mossin. Higher-order value flow graphs. In *9th International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, 1997.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.

[Pal94] Jens Palsberg. Global program analysis in constraint form. In Sophie Tison, editor, *19th International Colloquium on Trees in Algebra and Programming (CAAP'94)*, volume 787 of *LNCS*, pages 276–290. Spinger-Verlag, 1994.

[Pie91] B. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, School of Computer Science, Carnegie Mellon University, Dec. 1991. Technical Report CMU-CS-91-205.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Principles of Programming Languages*, 1995.

[PS92a] J. Palsberg and M. Schwartzbach. Safety analysis versus type inference. Technical Report PB-389, DAIMI, Aarhus University, March 1992.

[PS92b] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43:175–180, 1992. Also available as Tech. Rep. DAIMI PB-404, Computer Science Department, Aarhus University.

[Reh95] Jakob Rehof. Polymorphic dynamic typing, aspects of proof theory and inference. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, 1995.

[Rey74] J. Reynolds. Towards a theory of type structure. In *Proc. Programming Symposium*, pages 408–425. Springer-Verlag, 1974. Lecture Notes in Computer Science, vol. 19.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, CA, Jan. 1995.

[Rit95] M. Rittri. Deriving dimensions under polymorphic recursion. In *Functional Programming Languages and Computer Architecture (FPCA)*, pages 147–159. ACM Press, June 1995.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3), September 1986.

[RSH94] T. Reps, M. Sagiv, and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report 94/14, DIKU, University of Copenhagen, Denmark, 1994.

[Ses88] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988.

[Ses89] Peter Sestoft. Replacing function parameters by global variables. In *Fourth International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London*, pages 39–53. IFIP and ACM, ACM Press and Addison-Wesley, September 1989.

[Ses91] P. Sestoft. *Analysis and Efficient Implementation of Functional Languages.* PhD thesis, DIKU, University of Copenhagen, Oct. 1991.

[Shi88] Olin Shivers. Control flow analysis in Scheme. *Sigplan Notices*, 23(7):164–174, July 1988. Sigplan Conf. Programming Language Design and Implementation, Atlanta, Georgia, June 1988.

[Shi90] Olin Shivers. Data-flow analysis and type recovery in Scheme. Technical Report CMU-CS-90-115, School of Computer Science, Carnegie Mellon University, March 1990. 38 pages.

[Shi91a] O. Shivers. Data-flow analysis and type recovery in Scheme. In P. Lee, editor, *Topics in Advanced Language Implementation*, chapter 3, pages 47–88. MIT Press, 1991.

[Shi91b] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

[Shi91c] Olin Shivers. The semantics of Scheme control-flow analysis. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991).* ACM, 1991.

[SRH95] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proceedings of FASE '95: Colloquium on Formal Approaches in Software Engineering*, volume 915 of *Lecture Notes in Computer Science*, pages 651–665, Aarhus, Denmark, May 1995. Springer-Verlag.

[Sta79] Richard Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, Juli 1979.

[TJ92] Yan-Mei Tang and Pierre Jouvelot. Control-flow effects for escape analysis. In *Proc. Workshop on Static Analysis (WSA), Bordeaux, France*, pages 313–321, Sept. 1992.

[TJ94] Yan Mei Tang and Pierre Jouvelot. Seperate abstract interpretation for control-flow analysis. In *TACS'94*, volume 789 of *LNCS*. Springer-Verlag, April 1994.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Portland, Oregon (this proceedings)*. ACM, ACM Press, Jan. 1994.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7'th International Conference on Functional Programming and Computer Architecture*, pages 1–11, La Jolla, California, June 1995. ACM Press.

[vB95] Steffen van Bakel. Intersection type assignment systems. *Theoretical Computer Science*, 151(2):385–435, 1995.

[Wad89] P. Wadler. Theorems for free! In *Proc. Functional Programming Languages and Computer Architecture (FPCA), London, England*, pages 347–359. ACM Press, Sept. 1989.

[Yan90] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceeding of the Symposium on Principles of Database Systems*, pages 230–247, New York, 1990. ACM Press.

Index

$\cong$, 93
$\sqsubseteq$, 93
$\vdash^{\leq}_{=}$, 38, 40
$\vdash^{\leq}_{n}$, 41
$\vdash^{\wedge}$, 135–137
$\vdash^{s}$, 14, 20
$\vdash^{ML}$, 88
$\vdash^{ML}_{n}$, 89
$\vdash^{fix}$, 111
$\vdash^{fix}_{n}$, 111

annotation, 13
argument-empty, 162

BV, 11

cable, 69
    argument, 72
    binding, 72
    body, 72
    function, 72
    result, 72
cable-path, 69
carrier, 69
close, 97
closing rules, 66
closure analysis, 52
compact, 113
constraint, 13
    conditional, 33
constraints, 42
constructor, *see* Constructors
constructor node, 62
Constructors, 14
consume, 15
context, 11
    non-standard, 149
cycle
    elementary, 187

destructor, *see* Destructors
destructor node, 62
Destructors, 14
discard, 148
discriminants, 189
dynamic types, *see* type systems, dynamic

edge
    backward, 69
    forward, 69
either, 147
environment, 20
erasure, 19
expression, 11
    pseudo, 10
    well-named, 11

$\mathcal{F}$, 14, 21, 39, 91, 112, 135
firstification, 209, 212
flow function, 14
    intersection, 135
    ML-polymorphic, 91
    polymorphic recursion, 112
    simple, 21
    sound, 15
    subtyping, 39
flow graph
    typed, 68
    untyped, 62
flow property, 13
flow schemes, *see* formulae, predicative second order
formulae

first-order, 85
predicative second order, 85
simple, 19
FV, 11

instance
generic, 93
halbstark, 93
lazy, 43
simple flow analysis, 24
subtype flow analysis, 43
interface, 212

judgement
subtype, 37

$\mathcal{K}^{\forall}(t)$, 86
$\mathcal{K}^{\leq}$, 38
$\mathcal{K}^{s}$, 19
$\mathcal{K}^{\wedge}$, 134
Kleene-Mycroft sequence, 122

label
neutral, 50, 102
on expressions, 12
labelling, 12

ML polymorphism
flow analysis, 87
standard types, *see* type systems, ML polymorphism

n-level, 74
nesting
depth, 74
n-level, 74
n-nested, 74

occurrence
negative, 10
of label, 50, 102
positive, 10

path, 62
$?^{+}$-$?^{-}$, 72
call, 189
legal, 78, 189
return, 189
well-balanced, 79
pre-flow-graph, 63
proper, 24, 43

qualifier, 85
quantifier, 87

recursive types, *see* type systems, recursive types
reduction system
non-standard, 149
standard, 12
result-empty, 162

$\mathcal{S}^{\forall}(t)$, 86
semantics
non-standard, 149
standard, 12
solve, 14
subject expansion, 156
strong, 154
subject reduction, 31, 60, 107, 128, 154
substitution
label, 13
term, 12
subtype, 39
sum types, *see* type systems, sum types

term, *see* expression
type
flow, *see* formulae
recursive, 200
standard, 10
type system
dynamic, 202
ML polymorphism, 193
recursive types, 200
sum types, 197
typing
dynamic, 202
principal, 15, 28, 50, 101

unify, 22

value, 151
variable
    bound, 11
    free, 11

wbp, *see* path, well-balanced
well-named, *see* expression, well-named