Analysis and Efficient Implementation of Functional Programs

Peter Sestoft†

DIKU, Department of Computer Science, University of Copenhagen

October 3, 1991

(†) New address from January 1, 1992: Department of Computer Science, Building 344, Technical University of Denmark, DK-2800 Lyngby, Denmark. E-mail: sestoft@id.dth.dk

Preface

This report is a thesis submitted in partial fulfillment of the requirements for the Danish Ph.D. degree ("licentiatgrad"). It was written in the period April 1989 to October 1991 under the supervision of professor Neil D. Jones at DIKU, the Department of Computer Science, University of Copenhagen.

Guide for the reader

The report will hopefully be useful to people interested in functional programming languages, their implementations, and semantics-based analysis of functional languages. The reader should be familiar with functional languages and program analysis, but does not need to know much about their theoretical underpinnings as our approach will be quite operational throughout.

The main contribution of the report is the presentation of a number of automatic program analyses and transformations for functional programming programs. Some of these analyses are presented together with criteria for and proofs of their correctness, implementations of the analyses and transformations, or experiments made with the implementations.

The purpose of these automatic analyses is to collect information which can be used in subsequent optimizing transformations. Therefore the analyses collect *intensional* information, that is, information about the possible *executions* of a program, not just about the input-output function computed by the program.

One point illustrated is that the correctness of an intensional program analysis should be defined with respect to semantics which contain much operational information, or even with respect to abstract machines or abstract implementations of the language. In other words, the semantics must describe (aspects of) the intended implementations.

Acknowledgements

This thesis owes much to the collaboration with Carsten K. Gomard on a range of topics, documented in part below. In particular, Chapters 7 and 6 present joint work with Carsten, and some of that text will appear in his forthcoming thesis also. Working together with Carsten has always been highly productive and/or fun.

My former office-mate Hans Dybkjær (now at Roskilde University Centre) provided useful comments on chapters of this thesis as well as on many other notes, papers, and reports of mine over the years. Hans produced a package of syntactic extensions, which made my Scheme programming much easier.

More generally, the "TOPPS" or "Semantics-Based Program Manipulation Group" at DIKU and its head Neil D. Jones provided a stimulating environment with many visitors, travel, and new impulses and challenges.

It is a special pleasure to acknowledge the considerable role my friend Harald Søndergaard (now at Melbourne University) has played in my education as computer scientist.

A stay at Glasgow University with the functional programming group in the autumn of 1989 introduced me to lazy languages and their implementation. Warm thanks are due to Guy Argo, John Hughes, Thomas Johnsson, John Launchbury, and Phil Wadler for interesting discussions and hospitality. In particular, the work in Chapter 5 was inspired by discussions with Guy Argo on his improved Three Instruction Machine design.

Although not directly a member of the Semantique ESPRIT Project, I have benefited from the close contacts with Glasgow University, Imperial College, London, and Ecole Polytechnique, Paris.

The writing of this thesis has been accompanied by the usual crises and frustrations. Inevitably, some of them were caused by my supervisor, but Neil would always be ready with helpful and supportive comments and advice.

Last but not least, I want to thank my late mother Kirsten, my father Jørgen, and my dear Lone for their support.

This work was supported by Statens Naturvidenskabelige Forskningsråd (The Danish Natural Science Research Council) via Ph.D.-grant 11-7474.

Contents

Chapter 1

Introduction

This thesis deals with semantics-based analysis and transformation of strict and lazy purely functional languages. My goal is to explore ways to improve the implementation of functional languages via automatic program analysis and transformation. The improvement would be reduced run time consumption or storage consumption.

Although we shall construct an experimental implementation of a lazy functional language, the goal is *not* to construct a language implementation which is as efficient as possible in absolute terms. Rather, it is a study of some bricks that might be used in the construction of future language implementations or analyses and transformations.

We present a number of new analyses and transformations for strict and lazy languages. Some of these analyses have been implemented and proven correct, and some of the optimizing transformations have been tested on experimental implementations.

We also discuss notions of correctness of analyses and transformations. In particular, we discuss the relevance of proving them correct with respect to operational semantics or abstract machines, as opposed to denotational semantics or instrumented denotational semantics.

This chapter introduces some central terms and concepts.

1.1 Analysis and transformation of programs

By analysis and transformation of programs we shall mean automatic processing of the program text with the goal of obtaining either some information about the program (analysis), or a new program (transformation). Analyses are usually described by equations, but the goal typically is to construct programs for realizing the analyses. Thus the analyses should be effective, or computable, so that there are algorithms for solving the equations.

Program analysis produces some information about a program by analysing its text, without (full) information about the run time inputs to the program. As a consequence, such analysis information is almost always approximate for computability reasons. The analysis information we want to collect is often employed in a subsequent transformation of the program. Another use of analysis information is to show the programmer that he has expressed his intentions inadequately, by pointing out possible type errors, reporting bad binding time properties, estimates of run time consumption, *etc.* Program analysis in this sense is often called *compile time analysis* or *static analysis*.

Program transformation produces a new program which must be equivalent to the original one, at least under certain conditions. Often information from a preceding program analysis is used when checking such conditions for a given program. Thus automatic program analysis supports automatic program transformation.

A *semantics* for a programming language is an assignment of a meaning to every program in the language. A semantics can be considered a formal model of the language or certain aspects of it. (By a slight abuse, we shall sometimes use the phrase "the semantics of a program" for its meaning). A language may have a number of different semantics, varying in abstractness or the amount of information they give for a given program. Different semantics may reflect different kinds of implementations, but they should all agree on the more abstract aspects of the language. Traditionally, a programming language semantics is made as abstract as possible. Thus the meaning of a (functional) program may be taken to be the input-output function computed by the program, abstracting away the order of evaluation of subexpressions, run time consumption, *etc.* We shall use the word *extensional* for a semantics that prescribes only the input-output function, and *intensional* for semantics that include also other information.

An analysis or transformation is *semantics-based* if the result of applying it to a given program bears some formal relation to the meaning of the program in an appropriate semantics. Semantics-based analyses and transformations can thus be proved correct or wrong. This is particularly relevant in the present work, where we analyse programs for their possible run time behaviours. These behaviours are (usually) not prescribed by the standard denotational semantics. Thus an analysis or transformation may need to be related to a non-standard, non-abstract semantics to meaningfully characterize its correctness.

For example, the closure analysis presented in Chapter 4 computes a description of the set of functions an expression can evaluate to. Since the extensional semantics abstracts away the identity of functions, it does not contain sufficient information to say what it means for the closure analysis to be correct.

1.2 Functional languages: strict and lazy

A functional programming language is one in which the programmer relies mainly on (higher order) functions for expressing his intents. Functional programming languages can be classified along several dimensions: pure/impure, strict/lazy, typed/untyped, *etc.*

A *pure* functional language (*e.g.*, Miranda™, Haskell, or Lazy ML) is one in which side effects and global state are completely absent; an *impure* language (*e.g.*, Scheme, Lisp, Standard ML) allows side effects, possibly in restricted ways. Here we consider only pure functional languages.

A language is *strict* (or, *call by value*) if function arguments are always evaluated (exactly once) before the called function's body, and it is *non-strict* otherwise. A non-strict language may use *call by name*, in which an argument expression is evaluated every time it is used (never, once, or many times), or *call by need*, in which an argument expression is evaluated only at its first use (never or once). The distinction between call by name and call by need is operational (or intensional): switching from one to the other affects only the run time, not the result of a computation. The distinction between call by value (strict) and call by name or call by need (non-strict), on the other hand, affects the result: a computation may terminate under non-strict evaluation and fail to under strict evaluation.

A language has *lazy data structures* if the arguments of a constructor are evaluated by need, that is, at most once. A language that uses call by need and has lazy data structures shall be called a *lazy language*. In lazy languages (Miranda™, Haskell, or Lazy ML) it is rather hard to reason about the order of evaluation of subexpressions. This in turn makes it hard to reason about side effects, so lazy languages are usually pure. Strict non-toy languages (such as Lisp, Scheme, or Standard ML) are usually not. Here we work with both strict and lazy languages, and we shall see that they present different opportunities for optimization and pose different problems for analysis.

The pragmatic advantages of using lazy higher order functional languages are convincingly argued by Turner [71] and Hughes [39].

1.3 Abstract machines and implementations

The functional languages studied here will be described formally by a translation to suitable abstract machines. An *abstract machine* is a transition system which essentially resembles a real implementation without including irrelevant detail. Therefore abstract machines are useful for studying properties of implementations. We shall argue later that abstract machines may be more relevant than standard semantics or instrumented semantics in this respect.

1.3.1 Strict languages

The first abstract machine for evaluating a strict functional language is Peter Landin's SECD machine for the strict lambda calculus, which dates back to 1964 [44]. Real implementations of *e.g.* statically scoped Lisp can easily be based on the SECD machine as shown by Henderson [32]. A more recent example is Cousineau and Curien's Categorical Abstract Machine which is used in implementations of CAML, a variant of ML [12]. We shall later use a variant of the SECD machine when proving globalization analysis for a strict language correct in Section 7.7.

Modern real-life implementations of strict statically scoped languages such as Scheme and Standard ML are more sophisticated, though, and do not resemble the mainly interpretive implementations based on the SECD machine. They are based on converting the functional program into so-called continuation-passing style, which makes the flow of control more explicit and facilitates translation to traditional machine code. This approach was pioneered by Steele [66].

1.3.2 Non-strict languages

Abstract machines or evaluation mechanisms for non-strict or lazy languages are more recent, and seem to fall in two groups: graph reduction based and closure based.

Graph reduction machines directly achieve laziness by letting variables refer to pieces of a graph at run time. Replacing a subgraph with its reduced equivalent automatically affects all future uses of variables referring to it. The first graph reduction machine is due to Christopher Wadsworth and is exploited in Turner's combinator implementation of SASL [70]. Another graph reduction machine is the G-machine developed by Augustsson and Johnsson for their implementation of Lazy ML [41]. The G-machine is the focus of the most comprehensive textbook on implementation of lazy languages, Peyton Jones [51].

Closure based machines are basically non-strict (call-by-name) versions of SECD-type machines. That is, they are not necessarily lazy: they implement call by name instead of call by need. Laziness is achieved by *explicitly* introducing update actions into the machines. Closure based machines include Fairbairn and Wray's Three Instruction Machine [23] [74] and Krivine's machine (see *e.g.* [16] [15]) which are very similar although developed independently of each other. Both can be used directly as implementations of non-strict and lazy languages. We use a variant of the Krivine abstract machine for the semantics as well as implementation of a lazy example language in Chapter 3.

In graph reduction machines it is assumed that all subexpressions are shared; overwriting a graph with its reduced equivalent is the default mode of operation. In closure based machines it is easier to leave out an update operation if it is not needed, so they allow some optimizations for unshared expressions which graph based machines do not.

Modern real life implementations of lazy languages are hybrids of closure based and graph reduction based machines. One example is the Spineless Tagless G-machine Peyton Jones's [53] [52].

1.3.3 Run time and storage consumption

Two of the optimizing program transformations presented later are intended to reduce the run time consumption, primarily by reducing the number of actions needed to manage sharing in lazy languages (Chapters 5 and 6). The globalization analysis aims at reducing the storage consumption, and thereby also the run time consumption (Chapter 7).

Most functional languages (at least the lazy or higher order ones) require dynamic heap storage and thus a garbage collection mechanism. We shall assume that garbage collection methods such as "mark-and-sweep" or "stop-and-copy" are used [51, Chapter 17].

The use of heap means that (at least) two kinds of storage consumption must be distinguished. One is the *cell turnover*: the amount of allocation actions done during computation. The other is the *residency*, or live set: the number of cells live at some point in time. All else equal, a big cell turnover means that garbage collection will be frequent because the pool of free cells is used up rapidly, but each garbage collection may be fast (if the residency is small). Similarly, a big residency means that garbage collection will be frequent because the pool of free cells is small, and each garbage collection will be slow because all live cells have to be marked or copied (depending on the type of garbage collector). The maximal residency of a computation is important too: a program can run only when given at least as much storage as its maximal residency.

Reducing the cell turnover or the residency will save run time as well. Allocating a cell takes time for bounds checking, incrementing pointers, and other management. More importantly, frequent or slow garbage collections cause a considerable run time overhead in itself.

The analyses presented in this report primarily aim at reducing the cell turnover and the overhead of managing sharing, not at reducing the residency.

1.4 Optimized implementation

Here we present the setting for our work on analysing and implementing functional languages.

1.4.1 Why program analysis

Program analysis for efficient implementation is particularly pertinent to functional programming languages. One reason is the mismatch between the concepts of higher order functional programming languages and those of the (real) machines for executing them. The power and flexibility of functional languages stem from the fact that the phrases of a functional language can be combined freely (subject only to type restrictions). This flexibility is expensive; it means that a phrase must be implemented by a very general mechanism. However, in any given program it may be used only in a restricted way, so that a specialized and more efficient implementation is possible. Program analysis can discover such restricted uses in a given program and find the places where a specialized implementation will suffice.

This is true also for more conventional procedural languages, but the problem is less acute. Languages such as Algol, C, Pascal, or Ada make more concessions to the (real) machines on which they are implemented. Moreover, these languages contain a variety of special constructions which match machine concepts well, and which are likely to be used by the experienced programmer for efficiency. In a pure functional language, the programmer has to use the more general construct. The advantage is that he does not have to think about the most (machine) efficient way to express himself; the drawback is that he has to rely on the implementation to discover that he uses a general mechanism in a restricted way.

Thus a prominent purpose of analysis information is to identify cases of restricted use which allow a less general and more efficient implementation. Strictness analysis is a prime example: When a function argument is known to be evaluated anyway, one might as well evaluate it early and avoid the overhead of delaying and sharing its evaluation. This report also contains several examples of this pattern. The usage interval analysis finds function arguments which may be treated as unshared without loss of laziness (Chapter 5). The interference analysis finds function parameters which may be allocated globally (instead of using stack) without changing the meaning of the program (Chapter 7).

The kind of analysis information needed for improving (specializing) an implementation must tell us something about *all possible executions* of a program: What can safely be assumed about the use of a certain program phrase in all possible executions. In general an analysis providing such information must be proven correct with respect to a language semantics incorporating much intensional (or, operational) information. One exception is strictness analysis whose correctness can be proven on the basis of the (extensional) standard denotational semantics and within the framework of denotational abstract interpretation [47].

1.4.2 Transformation or implementation

In this work we shall use analysis information to guide optimizations not at the source level but at the underlying implementation level. One reason is that it is hard to understand the run time and storage consumption of a functional program by looking at the source program text itself. This problem is more pronounced for lazy languages than for strict ones because the order of evaluation is more complex. A consequence is that a transformation which preserves the meaning of a program may drastically (and unexpectedly) change its operational behaviour and its run time or storage consumption. For example, common subexpression elimination may cause a thousand-fold increase of the maximal residency and in the time spent in garbage collection, thus offsetting any run time saved by avoiding recomputation, cf. [51, p. 405].

Thus the effects of source level program transformation in lazy languages can be quite unpredictable. To get better control over the effects of transformations, the transformations and optimizations will be done on the abstract machine code. This also allows much finer optimizations: a source language construct is usually compiled to several abstract machine instructions, some but not necessarily all of which can be optimized away.

Thus our approach is "analysis plus optimization of the implementation" or "analysis plus target level transformation" rather than "analysis plus source level transformation". Moreover, the analyses we work with collect operational information: information about the possible executions of programs.

1.5 Plan of the report

The purpose of the *first part* of the report (Chapters 1 and 2) is to outline broadly the topics dealt with. It gives a general discussion of analysis and transformation of functional programs, and attempts to establish the meaning of "semantics-based".

The main *second part* presents several example languages, their operational semantics, and various analyses of these languages.

A higher-order untyped lazy example language L, and its operational semantics via compilation to an abstract machine called K are presented in Chapter 3. Then two analyses of L programs are presented: closure analysis (Chapter 4) and usage interval analysis, as well as an optimizing compilation of L programs (Chapter 5). Experimental implementations of the K machine, the analyses, and the compilation are listed and described in Appendix B.

A first order typed lazy language and an evaluation order analysis for lazy data structures in this language are presented in Chapter 6.

Finally, a strict language H and its operational semantics are presented as background for an interference analysis and a globalization transformation in Chapter 7.

The *third part* is the conclusion which summarizes what has been done and outlines topics for further investigation (Chapter 8).

The final *fourth part* contains the appendices: a list of terminology, program listings, a summary in Danish, the bibliography, and an index.

Chapter 2

Semantics-Based Program Analysis

We are interested mainly in analyses which provide information about the possible executions of a program. Such analyses are said to be semantics-based, because they relate to the meaning or denotation of a program, not to the shape of its text, for instance. The present chapter discusses this in more detail.

2.1 Semantics and program analysis

A *base semantics* specifies the "standard" required behaviour of a program when executed. A semantics may be given in the style of denotational semantics (Strachey [69]), structural operational semantics (Plotkin [55]), natural semantics (Kahn [43]), or equational algebraic semantics [20].

A semantics may also be given as a translation to some other well-understood language. A special case is the translation to the language of an abstract machine (such as the SECD or Krivine machines used in this report), which in turn is described (operationally) by a transition system or by other means. Indeed, a denotational semantics may be considered a translation from the source language being defined into lambda calculus terms, usually involving recursive equations or a fixed-point operator. The meaning of lambda terms can then be given either by the reduction rules of the lambda calculus, or by interpreting lambda terms as elements of suitable domains, thus ultimately assigning a meaning to the terms of the source language.

An advantage of translation to an operationally defined abstract machine language is that it makes the semantics more operationally *plausible* by basing it on something akin to a real machine implementation, instead of the somewhat complex lambda calculus models in denotational semantics.

2.2 Abstract interpretation

Abstract interpretation is a special methodology for program analysis. Every operator in the language is reinterpreted by a corresponding "abstract" operator which mimics the effect of the original "concrete" one. Under suitable assumptions the evaluation of a term with the abstract operators will give an (abstract) result which correctly mimics the (concrete) result of an ordinary evaluation.

A simple example is "calculation with signs". We replace all negative numbers with the abstract value "$-$" and all positive numbers with "$+$", and replace multiplication of numbers by the "signs rules", such as $+ * + = +$ and $+ * - = -$, $- * 0 = 0$, *etc.*

A classical example in program analysis is strictness analysis. The value domain is replaced by the two-point domain $\{0,1\}$ where 0 means "undefined" and 1 means "possibly defined", and all operators are reinterpreted accordingly. For example, $+0 = 0$, $0+1 = 0$, $1 + 0 = 0$, and $1 + 1 = 1$ say that $+$ is strict in both its arguments, while `if` $011 = 0$, `if` $110 = 1$, and `if` $101 = 1$ say that `if` is strict only in its first argument.

Many program properties can be calculated by choosing suitable abstract value domains and abstract operators, and evaluating the program abstractly. It is important to choose the abstract domains and operators so that abstract evaluation will always terminate.

Abstract interpretation was first described by Sinztoff [64], although Naur used the idea for type checking already in [48]. Cousot and Cousot formalized and developed the idea [13], Mycroft applied it to functional languages for strictness analysis [47], and Nielson developed denotational abstract interpretation [49]. This summary draws on Søndergaard's thesis which gives a good account of the development of abstract interpretation [65, Chapter 3].

2.3 Semantic analysis information

We shall distinguish several kinds of (analysis) information about a program. These kinds can be structured in levels by analogy to classical compiler terminology, using the terms lexical, syntactic, and semantic [2].

On the first level we have *lexical information* , such as the length of the program text in lines, the number of occurrences of the letter "P" in the program text, *etc.* On this level, a program is considered a stream of characters, and a concept such as variable name makes no sense.

On the second level we have *syntactic information*, such as the number of different variable names appearing in the program, the deepest lexical nesting of scopes, *etc.* On this level, a program is considered an abstract syntax tree, and concepts such as lay-out and the length in characters of a function definition make no sense.

On the third level, we have *semantic information*, such as the set of input data for which the program terminates, the set of possible outputs for inputs which are even numbers, or the maximal run time for inputs of size less than n, *etc.* On this level, a program may be considered a function from input values to output values, or a function from input values to run time consumption (in reduction steps or microseconds), *etc.*

As these examples of semantic analysis information may suggest, it is useful to make an even finer distinction. Thus we shall distinguish between *extensional* and *intensional* semantic analysis information. Extensional information is concerned purely with the input-output behaviour of an entire (functional) program, whereas intensional information may include other things, such as the order of evaluation of subexpressions, the number of uses of a variable, the input-output behaviour of components of the program, or the run time needed to execute the program. We will also use the term *operational* for intensional semantic information.

For an example of intensional information, consider complexity theory [25]. In complexity theory, run time and space consumption are usually treated as properties of algorithms, but only with respect to a very precise class of implementations or computational models, such as Turing machines.

In our view, a *semantics-based* analysis is one which finds approximate semantic information about a program. A semantics-based analysis can be further classified as *extensional* or *intensional*, depending on the sort of information it finds. An extensional semantics-based analysis finds information about the input-output function computed by a program; an intensional semantics-based analysis finds information about the possible executions of the program.

Sometimes extensional information is identified with denotational semantics, and intensional information with operational semantics. This identification can be misleading, since intensional information may well be specified using denotational semantics. A denotational semantics could for example specify the run time consumption as well as the result of execution for a program.

One reason for identifying extensional and denotational is that a denotational semantics frequently is used to provide a standard for the language defined, giving only extensional information. For example, when Stoy says that denotational semantics should be used provide a *normative* description of a language [69, p. 24], this may leave the impression that it cannot be used in other modes (such as the descriptive) also.

One reason for identifying intensional information and operational semantics is that although operational semantics can be used normatively, it is likely to include more "accidental" detail than a denotational semantics, such as the order and number of transitions needed to run a program.

2.4 Instrumented semantics

This report deals mainly with intensional semantics-based analyses. An analysis collecting such information clearly cannot be proved correct with respect to a semantics not already containing it. In particular, it cannot be proved correct with respect to a denotational base semantics.

One way to solve this problem is to use an *instrumented semantics* or *extended semantics*: a semantics which prescribes some of the operational aspects of a program in addition to its base semantics. The abstract interpretation paradigm can now be applied to the instrumented semantics to obtain approximate operational information which is correct with respect to the instrumented semantics. Often the instrumented semantics is obtained from the base semantics by adding extra components to the denotation of an expression. Examples of this technique include Hudak's reference count semantics [34], Bloss and Hudak's path semantics [8], Goldberg's sharing semantics [27], Park's reference escape semantics[1] [50], and Shivers' "non-standard abstract semantics" [61] [62].

2.5 Correctness of analyses

We shall see that in general analysis correctness must be defined with reference to semantics which contain more operational detail than the "usual" denotational semantics, which prescribes only the input-output behaviour of programs.

A program analysis works on a program text without (exact) information about the run time input to the program. Thus for reasons of computability, semantics-based program analyses are almost always approximate. Rice's theorem states that "Any nontrivial property $\mathcal{S}$ of the recursively enumerable languages is undecidable" [33, p. 188]. This result implies that (non-trivial) semantic properties of a program cannot be decided from the program text (by a mechanical procedure). Certain properties of program texts and their computations *are* decidable, but even so it will frequently happen that the property detected by a program analysis is really an approximation to some undecidable property.

This means we cannot require program analyses to be *exact*. Then what does it mean for a program analysis to be correct? A simple and utilitarian requirement is that the analysis gives information which serves its purpose, as when the information is used in a subsequent program transformation.

For instance, the transformation may preserve the meaning of the program only in case the program has certain properties, and we want the analysis to decide whether it has. The analysis must provide a *safe* answer. The analysis should say that the program has the required properties only if this is actually the case. Otherwise we may wrongly apply the subsequent transformation to a program for which it is not valid. That is, we accept a conservative ("harmlessly wrong") answer, as long as it errs on the safe side. It *must* prevent us from invalidly applying the transformation, and it *may* prevent us from validly applying the transformation.

2.5.1 Joint correctness of analysis and transformation

Thus we might suggest an *extensional* criterion for analysis correctness based only on the input-output behaviour of programs. The program analysis is correct if using its results in the transformation makes it preserve program input-output behaviour.

We may formalize this notion of analysis correctness as follows. Let S and L be programming languages. Let t be a program transformation which takes an S-program s and some information i about s as input and produces an L-program l. Provided that the information i about s is correct, l should be equivalent to s, that is, $Ll = Ss$. Further assume that we have a program analysis a which when given s should produce the information i about s.

[1]Common to these works is that their instrumented semantics are not presented as conservative extensions of the denotational base semantics. Instead of computing and passing along precise values, an "oracle" is used to decide the conditional, thus obtaining something which is already somewhat approximate. However, it is rather straightforward to add in the details necessary to obtain a conservative extension of the denotational base semantics.

Then we may define that the analysis a and the transformation t are *jointly correct* if for all $s \in S$-programs, $Ss = Ll$ where $l = t(s, a(s))$.

For example, the information i may be strictness information about function parameters; a may be a strictness analysis for S-programs; and t may be the transformation that makes l evaluate arguments in strict positions *before* function calls. Indeed, the transformation is meaning-preserving only if the strictness information is correct. If the strictness analysis is wrong, the transformation may move the evaluation of a non-terminating argument ahead of a function which does not need it. This may change a terminating program into a non-terminating one, thus corrupting the input-output behaviour of the transformed program.

2.5.2 Joint correctness is not sufficient

Joint correctness seems to be the most general extensional analysis correctness criterion. But there are situations where joint correctness is not sufficient to characterize correct analyses. It may be the case that "wrong" analysis information will affect only the *efficiency* or *size* and not the input-output behaviour of the transformed program l. Then the joint correctness criterion fails to capture the wrongness of the analysis.

For an example of this, i may be information about the sharability of argument expressions and t may be the transformation from call by need to call by name for all expressions that are unshared according to the information i (see Chapter 5). If the analysis a classifies an expression as unshared when it actually is shared, then the transformed program may run orders of magnitude too slowly, but will still produce the expected output. The reason is that if the update mechanism is (wrongly) left out for an expression which is in fact shared, then the expression will be evaluated with call-by-name instead of call-by-need. Hence the analysis is wrong if it classifies too few expressions as shared, but the extensional joint correctness criterion does not see the difference.

In conclusion, the joint correctness requirement is too weak in some situations: It puts requirements on the extensional aspects of a program but leaves the intensional properties (such as run time and storage consumption) unspecified. This thesis deals to a large extent with analyses for which the joint correctness requirement is insufficient. Below we shall discuss other notions of correctness, in particular correctness with respect to instrumented semantics.

2.5.3 Correctness of abstract interpretation

One of the great attractions of abstract interpretation is that when the abstract domains and operators satisfy certain requirements, the standard theory guarantees that the result of the abstract interpretation is correct with respect to the abstracted semantics.

At first it seems natural to require an abstract interpretation to be an abstraction of the denotational base semantics. In fact, this requirement may be responsible for the considerable interest in strictness analysis. Strictness analysis is one of the few semantics-based analyses which can readily be justified with respect to the denotational base semantics. That it can, is due to the special interpretation (as non-termination) of the least element

$\perp$ of the domains in which the lambda terms of denotational semantics are modelled.

However, in general one cannot find intensional analysis information by abstract interpretation of a denotational base semantics. The base semantics does not contain sufficient information.

2.5.4 Abstract interpretation of instrumented semantics

One solution is to do abstract interpretation of an instrumented semantics, usually an extension of the denotational base semantics. The denotation of *e.g.* an expression contains some additional information besides the "value" or effect of the expression. For instance, this additional information may be a sequence of variables in the order they would be accessed during evaluation.

Abstract interpretation of instrumented semantics is the approach to program analysis taken in particular by the Yale group led by Paul Hudak.

For example, Hudak derives a reference count analysis from his (instrumented) reference count semantics [34]. Similarly, Bloss derives a path analysis (essentially an evaluation order analysis) from her path semantics [7]. Goldberg derives a sharing analysis from his sharing semantics [27], and Park and Goldberg derives a reference escape analysis from their escape semantics [28] [50]. As another example, Shivers derives a control-flow analysis from an instrumented semantics for Scheme which incorporates information about functions [60] [63]. This is similar to our closure analysis, but ours is proven correct with respect to an abstract machine (see Chapter 4).

Thus abstract interpretation of instrumented semantics gives a method for stating and proving correctness of intensional analyses in a way that closely resembles the way extensional analyses can be proved correct with respect to the denotational base semantics.

2.5.5 Correctness of instrumented semantics

This approach raises the question of correctness of the instrumentation. The purpose of a base semantics is to prescribe the correct input-output behaviour of programs, so the instrumented semantics should certainly agree with the standard semantics on input-output behaviours. This can be ensured by choosing the instrumented semantics as a conservative extension of the standard semantics. However, this does not prove that the instrumented semantics and thus the analysis is correct or useful in any way. It only proves that it is possible to find a machine or implementation for which they would be correct.

But when should the extensions (over base semantics) be deemed correct? This depends on the way one uses instrumented semantics. There are two possibilities:

First, like the base semantics it may be used in a normative manner, thus characterizing the valid implementations of a language. This way it is correct by definition, and implementations should be derived from or proved correct with respect to the instrumented semantics.

Secondly, an instrumented semantics may be used as a concise and sufficiently abstract way to summarize the properties of a reasonable implementation. This way the instrumented semantics should be derived from or proven correct with respect to the implementation.

Moreover, it is always a good idea to have an explicit model of the phenomenon (*e.g.*, reference counts) under investigation. In this respect the situation is the same as for formal specification of programs using formal specification languages such as Meta-IV, Z, *etc.* The important effect of (careful) formalization is that one is forced to work through all aspects and details of the problem at least once. Tacit assumptions are discovered and made explicit, one's consciousness about the subject matter increases, and mistakes are spotted more easily. Maybe this is the primary pragmatic benefit of an (instrumented) semantics, since implementations are seldom proved correct with respect to the instrumented semantics or vice versa.

A small problem with instrumented semantics lies in the connotations of the term itself. As noted above, a base semantics is frequently used in a normative or prescriptive manner, which makes it pointless to question its correctness. Therefore the term "semantic" is often used with the connotation "correct by definition". This is problematical with instrumented semantics: the positive connotations of "semantics" may lead the researcher to forget that his instrumentation may be quite arbitrary.

2.5.6 Abstract machines

Proving an analysis correct with respect to an abstract machine avoids questions of this kind, in particular if it is possible to build an implementation similar to the abstract machine. This may require more *ad hoc* methods than those of the abstract interpretation paradigm, but it removes the need to postulate an instrumentation, and to prove the instrumentation correct or reasonable with respect to an implementation.

When using the analysis results for improving program implementations, this has the further advantage that the intensional analysis results are guaranteed to be applicable in implementations closely modelled on the abstract machine.

Chapter 3

A Lazy Example Language

Here we introduce a lazy example language to be used in later chapters. A language must have a name, so we shall call it L for lazy. We also give a simple abstract machine called K to execute it; it is a variant of the so-called Krivine machine. The K machine will be used as the operational model with respect to which we prove our analyses and transformations of lazy languages correct. In the chapter on globalization for partial applications in a strict language, we shall use a version of Landin's SECD machine, not the Krivine machine, for the operational semantics.

3.1 The lazy language L

The language L evaluates function arguments by need and has lazy data structures, in the manner of other lazy languages, such as Miranda™, Lazy ML, and Haskell.

Thus the language has non-strict application and non-strict constructors. It is evaluated using what might be called "*weak head reduction*" or "*outermost weak reduction*" or "*reduction of top level redexes*"[1]. However, this does not fully characterize the intended implementations. In addition, the implementation must be lazy so the argument to a function application is evaluated at most once for each evaluation of the application, and the arguments to a constructor application is evaluated at most once for each evaluation of the constructor application.

3.1.1 Syntax and informal semantics

The syntax of the L language is given below. We shall start by implementing only its *applicative kernel*: abstraction, variables, and application. Then we extend it with base values, conditional, data structures, and local recursive definitions.

[1] I have not found an authoritative source for these terms. Peyton Jones calls this "normal order reduction of top-level redexes" [51, p. 198], Turner calls it "normal order reduction" [70, p. 41], and Fairbairn and Wray calls it "normal order evaluation" [23, p. 35], but it is *not* leftmost reduction as in Barendregt [5].

e	::= λx.e	Lambda abstraction
	| x	Variable or function
	| (e e)	Application
	| A	Base function
	| c	Base constant
	| if e_1 e_2 e_3	Conditional
	| Nil	Nil constructor
	| Cons	Cons constructor
	| { case e of Nil => e_1 (Cons x xs) => e_2	Case analysis for data structures
	| letrec def* in e_0	Local recursive definitions
def	::= f x_1 ... x_n = e	Variable or function definition $n \geq 0$

The intended semantics of this language is like that of *e.g.* Lazy ML or Haskell. Thus application of lambdas is non-strict and lazy. In the application ((λx.e_1) e_2), the argument expression e_2 is never evaluated if x is not accessed during evaluation of e_1, and it is evaluated at most once regardless of the number of accesses to x in e_1.

Application of base functions A is strict, and if e_1 ... and case e_1 of ... are strict in e_1. For case, this means that e_1 is evaluated sufficiently to find out whether the constructor is Nil or Cons.

Constructors in data structures are non-strict and lazy too. The arguments to Cons are not evaluated unless they are needed in the right hand side of a case rule, and they are evaluated at most once regardless of the number of uses.

The local recursive letrec definitions are also non-strict and lazy. The right hand side e_i of a binding x_i=e_i is not evaluated at all if x_i is not used in the letrec-body e_0 or in any right hand side which is evaluated. Moreover, e_i is evaluated at most once regardless of the number of uses of x_i.

For implementation and analysis purposes we shall use a sublanguage, the so-called *core L* language. The full L language will be explained by simple syntactic translations into the core language in the relevant sections below. The syntax of core L is

e	::= λx.e	Lambda abstraction
	| x	Variable
	| (e e)	Application
	| (A e_1 ... e_n)	Fully applied base function
	| if e_1 e_2 e_3	Conditional
	| letrec f_1 = e_1 ... f_m = e_m in e_0	Local recursive definitions

Note that the applicative kernel is part of core L. The main limitations of the core language are that base functions must be fully applied, that local recursive definitions must be argumentless, and that there is no explicit handling of data structures. The only kinds of reduced values in core L are functions λx.e and base values c.

3.2 The call by name Krivine machine

To implement core L we use a variant of the Krivine machine, which we shall refer to as the K machine. The Krivine machine is an abstract machine for call by name evaluation of lambda terms, invented by the French logician J.-L. Krivine around 1985 and described in *e.g.* [16] and [15]. The Krivine machine directly implements outermost weak reduction, or call by name evaluation to weak head normal form, for the applicative kernel of L. We shall not here go into a proof that the Krivine machine actually implements this reduction strategy.

The Krivine machine is similar to but simpler than Fairbairn and Wray's Three Instruction Machine TIM [23] [74]. Operationally speaking, the main difference is that it uses lists instead of arrays for the representation of environments, which simplifies the handling of shared closures in the lazy version of the machine, but slows down variable access. When extending the basic Krivine machine with base values, laziness, *etc.*, we shall borrow several of Fairbairn and Wray's ideas as used in the TIM.

The K machine will be described using state transition schemas. The machine state (`C`,E,S) has three components: The current code `C`, the current environment E, and a stack S. The code `C` is a K machine instruction, actually just a lambda term in disguise, and the environment E and the stack S are lists of values. A value u is a function, represented as a *closure* u = `C`•E consisting of *code* `C` and an *environment* E. (For now we ignore base values and base functions).

The code `C` uses deBruijn indices for variables; the environment E provides values u_0 : u_1 : ... for the (free) variables of `C`; and the values on the stack S can be considered arguments to the function represented by `C`•E. Alternatively, the stack top element may be considered the continuation (return address) for the code `C` in environment E, and the remaining stack elements as a dump in the sense of Landin's SECD machine [44].

The syntax and meaning of K machine instructions will be apparent from the defining rules below.

The assertion that from machine state (`C`,E,S) one may reach machine state (`C`′,E′,S′) in one step of computation is written (`C`,E,S) ⇒ (`C`′,E′,S′). The assertion that from machine state (`C`,E,S) one may reach machine state (`C`′,E′,S′) in zero or more steps of computation is written (`C`,E,S) ⇒* (`C`′,E′,S′).

The rules defining the basic call by name Krivine machine (without base values or data structures) are

Code C	Env. E	Stack S	⇒	Code C	Env. E	Stack S	
App C_1 C_2	E	S	⇒	C_1	E	C_2•E : S	
Var i	E	S	⇒	C_i	E_i	S	where C_i•E_i = E[i]
Lam C	E	v : S	⇒	C	v : E	S	

The pair `C`•E of a machine state (`C`,E,S) is called the *current closure*. We say that a closure is activated or entered when it is taken from the environment (as in the `Var` rule), or from the stack, and made the current closure. Activating a closure is tantamount to demanding its value, that is, demanding its weak head normal form (whnf).

The machine stops when no rule applies. This happens when an abstraction `(Lam C)` finds the stack empty, so that it cannot be applied. In this case the closure `(Lam C)`•E is the result of the computation. This is the reduced value or *weak head normal form* (*whnf*) of the closure the machine was started with. The environment provides bindings for the free variables of `(Lam C)`.

In the basic K machine all reduced values or whnfs have form `(Lam C)`•E. In the next section we add base values which are different.

3.2.1 Translation from L to K machine code

The translation of the applicative kernel of L into K machine instructions is just a change of notation from named variables to deBruijn notation. We shall give the remaining translation rules in the relevant subsections below.

$$
\begin{array}{lcl}
\mathcal{C}[\![\, \lambda \mathtt{x}\,.\,\mathtt{e} \,]\!]\rho_c & = & \mathtt{Lam}\ \mathcal{C}[\![\, \mathtt{e} \,]\!](\mathtt{x}{:}\rho_c) \\
\mathcal{C}[\![\, \mathtt{x}_i \,]\!][\mathtt{x}_0,\ldots,\mathtt{x}_i,\ldots,\mathtt{x}_n] & = & \mathtt{Var}\ i \\
\mathcal{C}[\![\, \mathtt{e}_1\ \ \mathtt{e}_2 \,]\!]\rho_c & = & (\mathtt{App}\ \mathcal{C}[\![\, \mathtt{e}_1 \,]\!]\rho_c\ \mathcal{C}[\![\, \mathtt{e}_2 \,]\!]\rho_c)
\end{array}
$$

3.2.2 Handling base values

To represent the base values of core L we extend the repertoire of whnfs in the K machine. We achieve this as in Fairbairn and Wray's TIM [23].

For instance, the number 17 is represented as the pseudo-closure `Ret`•17, and the Boolean value True is represented as `Ret`•T. In general, the representation of base value v is `Ret`•v, where v is a number or a Boolean. The code component of the closure simply says "this is a base value", and the environment component is (ab)used to hold the actual value v. The action of such a pseudo-closure is to swap the stack top element with itself. That is, it pops the stack top element, pushes itself, and activates the old stack top element. The stack top element therefore must be a function which expects a base value as argument. The action of `Ret` can be seen as a return jump from the computation of the argument to the function which needs it; hence its name[2].

The rule for (eliminating) base values and the rule for introducing them (by the instruction `Cst`) are

`Ret`	v	C•E : S	⇒	`C`	E	`Ret`•v : S
`Cst c`	E	S	⇒	`Ret`	c	S

Now there are two kinds of whnf in the K machine, namely `Lam C`•E and `Ret`•v, corresponding to the two kinds of reduced value in core L: functions and base values. If a `Ret` instruction finds the stack empty, no rule applies, and the machine stops. The result of the computation is the closure `Ret`•v, representing the base value v. Note that in the final machine state one can distinguish base value results (of form `Ret`•v) from function type results (of form `Lam C`•E).

[2]This is the name used by Simon Peyton Jones and Guy Argo. The name originally used by Fairbairn and Wray was `Self`.

To keep the number of concepts to a minimum, we might have used a functional encoding such as Church numerals for natural numbers and Booleans, but that is so inefficient as to make practical experiments with an implementation infeasible. Hence we have chosen the faster but more slightly more complicated representation just outlined.

Now that base values, including Booleans, have been added to the machine, we can implement the source language conditional `if` e_1 e_2 e_3 by adding an instruction `If` to the machine. It simply looks at the first element on the stack, which must be a Boolean, to decide whether to enter the second stack element (if True) or the third one (if False).

`If`	E	Ret•T:C_1•E_1:C_2•E_2:S	⇒	C_1	E_1	S
`If`	E	Ret•F:C_1•E_1:C_2•E_2:S	⇒	C_2	E_2	S

Base value constants and conditional expressions are compiled as follows:

$$\mathcal{C}[\![\,\texttt{c}\,]\!]\rho_c = \texttt{Cst c}$$
$$\mathcal{C}[\![\,\texttt{if } e_1\ e_2\ e_3\,]\!]\rho_c = (\texttt{App (App (App } \mathcal{C}[\![\,e_1\,]\!]\rho_c\ \texttt{If) } \mathcal{C}[\![\,e_2\,]\!]\rho_c\texttt{) } \mathcal{C}[\![\,e_3\,]\!]\rho_c\texttt{)}$$

Using this compilation, `if` becomes strict in the conditional expression (as one would expect), since the conditional expression is applied to the `If` instruction.

We consider only three cases of fully applied base functions (A e_1 ... e_n) in core L, namely, $n = 0, 1, 2$. We have already considered base constants $\texttt{c} \equiv \texttt{(A)}$, and shall now look at the implementation of fully applied monadic functions (A e_1) and fully applied dyadic functions (A e_1 e_2).

A monadic base function application (op e_1) is compiled to (App C_1 `B1-op`), where C_1 is the compiled code for e_1, and `B1-op` is the K machine instruction corresponding to base function `op`. This means that a monadic base function will always find its evaluated argument (of form Ret•v) on the top of the stack. The function just has to extract the base value v from this closure.

For a dyadic base function, some argument swapping is needed. The application (op e_1 e_2) is compiled as (e_1 `R1` e_2 `B2-op`). The role of `R1` is to rearrange the stack top elements and start evaluation of e_2, so that e_2 will eventually activate `B2-op`. Thus fully applied monadic and dyadic base functions are compiled as follows:

$$\mathcal{C}[\![\,\texttt{(A } e_1\texttt{)}\,]\!]\rho_c = \texttt{App } \mathcal{C}[\![\,e_1\,]\!]\rho_c\ \texttt{B1-A}$$
$$\mathcal{C}[\![\,\texttt{(A } e_1\ e_2\texttt{)}\,]\!]\rho_c = \texttt{App (App (App } \mathcal{C}[\![\,e_1\,]\!]\rho_c\ \texttt{R1) } \mathcal{C}[\![\,e_2\,]\!]\rho_c\texttt{) B2-A}$$

The additional K machine rules for `B1-op`, `R1`, and `B2-op` are

`B1-op`	E'	Ret•v : S	⇒	Ret	(op v)	S
`R1`	E'	Ret•v_1 : C•E : u : S	⇒	C	E	u : Ret•v_1 : S
`B2-op`	E'	Ret•v_2 : Ret•v_1 : S	⇒	Ret	(op v_1 v_2)	S

This implementation of base functions can be extended to work for strict triadic, tetradic, *etc.* base functions, by introducing appropriate new instructions such as `R2`, `R3`, ..., and `B3-op`, `B4-op`, ... The idea is that instruction Rm is executed after argument number m of a base function with arity greater than m has been evaluated. This instruction must move stack element $m+2$ to the stack top, and activate element $m+1$. An instruction of form Bm-op applies op to the m topmost stack elements, and enters the result. This implementation of base functions of any fixed arity is very regular, but probably rather inefficient for high arities because of all the stack rearrangements.

The reader may have noticed that with the new rules the machine could stop because an instruction such as If or B1-op finds an insufficient number of elements on the stack. However, this will not happen if the K machine code is compiled from fully applied base functions and well-formed if e_1 e_2 e_3 expressions.

3.2.3 Running the K machine

When evaluating an expression e, the run time inputs are given through its free variables. The expression is evaluated by the K machine starting from the state (C,E^0,[]) where C $= \mathcal{C}[\![\, e \,]\!]\rho_c$ is the code compiled for e, and E^0 is the initial environment with bindings for the free variables of C. The order of the variables in the initial compile time environment ρ_c is the same as the order of their values in E^0.

Applying the K machine rules to one state of the machine, the next state is obtained. In this manner, one can develop a (finite or infinite) *trace* of its execution from the initial state.

3.3 Adding data structures

In this section we extend the language L with data structures such as pairs, lists, trees, *etc.* We shall refer to these as algebraic data structures. Instead of extending the K machine with special constructs to handle data structures, we encode them as lambda terms. This has several advantages. First, when we make function application lazy, we get lazy data structures at no extra cost. Secondly, this keeps the number of instructions in the K machine low, which is a great advantage when doing proofs by induction on machine executions. Fairbairn and Wray also suggested to encode data structures as functions in the TIM [23]. We shall not go to the extreme of encoding numbers *etc.* as Church numerals: The resulting data structures will be vast and the loss of efficiency as compared to machine arithmetic too big.

The particular encoding we use is related to the encoding of products and sums as functions in *e.g.* Reynolds [56, p. 130–131] and Pierce *et al.* [54]. However, the encoding of recursive data structures is done by ignoring the recursive type structure as in Steensgaard-Madsen [68], so we do not obtain self-iterating encodings. This is appropriate since the L language will have recursion anyway (via letrec).

Thus we can add data structures by a modification of the compilation phase and need a small extension of the K machine only to cater for lazy printing of data structures.

We extend the language with Lisp-style S-expressions, and avoid the introduction of a proper type system. This creates some special problems for lazy printing of the result of a computation, discussed in Section 3.4.4 below. However, we first describe printing in the well-behaved typed case in Section 3.4.3. We have not implemented the typed theory to avoid constructing a type inference algorithm and a more complicated compilation algorithm.

3.3.1 Data structures encoded as lambda terms

In general, an algebraic *data type* t is defined as a disjoint sum of products, using the following syntax

$$t \ \texttt{::=} \ C_1 \ t_{11} \ \ldots \ t_{1c(1)} \mid \ldots \mid C_n \ t_{n1} \ \ldots \ t_{nc(n)}$$

Each C_i is a *constructor*, of arity $c(i)$. The arity may be 0, in which case C_i is simply a new constant. The constructors must be distinct. The t_{ij} are type expressions, which must be base types, or the name of some algebraic data type t. A type declaration of this form extends the syntax of the language with the n constructors C_i of type $t_{i1} \to \ldots \to t_{ic(i)} \to t$, and with a destructor `case` for case analysis on the value of expression `e` of type t:

$$\begin{array}{l} \texttt{case e of} \\ \quad C_1 \ \texttt{x}_{11}\ldots\texttt{x}_{1c(1)} \ \texttt{=>} \ \texttt{e}_1 \\ \quad \ldots \\ \quad C_n \ \texttt{x}_{n1}\ldots\texttt{x}_{nc(n)} \ \texttt{=>} \ \texttt{e}_n \end{array}$$

We call `e` the *root expression* of the `case`-expression, and refer to each of the n rules C_i $\texttt{x}_{i1}\ldots\texttt{x}_{ic(i)}$ `=>` $\texttt{e}_i$ as a *case* or *branch* of the `case` expression.

In the functional representation, a value v of type t is a function of arity n: the number of constructors of type t. Each constructor C_i is encoded as a function $\widehat{C_i}$ which takes $c(i)$ arguments and produces a value of type t. This value in turn is a function of n arguments, which will apply its i'th argument to the $c(i)$ arguments of $\widehat{C_i}$. Thus the encoding of constructor C_i is

$$\widehat{C_i} \ \texttt{=} \ \lambda\texttt{x}_1.\ldots\lambda\texttt{x}_{c(i)}.\lambda\texttt{c}_1.\ldots\lambda\texttt{c}_n.(\texttt{c}_i \ \texttt{x}_1 \ \ldots \ \texttt{x}_{c(i)})$$

The destructor `(case ... of ...)` for values of type t is encoded as the application of the root expression to n lambda terms, one for each branch of the `case` expression.

$$(\texttt{e} \ (\lambda\texttt{x}_{11}.\ldots\lambda\texttt{x}_{1c(1)}.\texttt{e}_1) \ \ldots \ (\lambda\texttt{x}_{n1}.\ldots\lambda\texttt{x}_{nc(n)}.\texttt{e}_n))$$

It is easy to see that this gives the desired behaviour. If the root expression of a `case` is $(C_i \ \texttt{e}'_1 \ \ldots \ \texttt{e}'_{c(i)})$, then the i'th branch $(\lambda\texttt{x}_{i1}.\ldots\lambda\texttt{x}_{ic(i)}.\texttt{e}_i)$ will be "selected" and applied to $\texttt{e}'_1 \ \ldots \ \texttt{e}'_{c(i)}$. Thus the result of the `case` is that the i'th right hand side $\texttt{e}_i$ is evaluated with $\texttt{x}_{ij}$ bound to component $\texttt{e}'_j$ of the data structure, exactly as intended.

3.3.2 Data structures in L

The handling (especially printing) of general data structures encoded as functions requires a type system. Since we are not interested in type issues here, we shall use a single data structure which is general enough to do interesting lazy functional programming, yet simple enough not to require type inference. We extend the syntax of the language to handle Lisp-like binary S-expressions, given by the type definition

S-expr = `Nil` | `Cons` value value

We extend the language with the following syntactic elements:

```
e ::= Nil                          Empty list constructor
    | Cons                         Pair constructor
      ⎧case e of
    | ⎨     Nil         => e1      Case analysis
      ⎩     (Cons x xs) => e2
    | hd                           Head element of list
    | tl                           Tail element of list
    | nl?                          Test for empty list
```

According to the general encoding scheme for data structures, we simply consider these as syntactic shorthands for core L expressions. In particular, the applied selectors `(hd e)` and `(tl e)` abbreviate `case`-expressions. We have arbitrarily decided that they return the empty list `Nil` when applied to `Nil`. As for base functions, a non-applied selector is eta-converted so that it becomes fully applied.

```
Nil                   is λn.λc.n
Cons                  is λx.λy.λn.λc.(c x y)
case e of          ⎫
     Nil      =>e1 ⎬  is (e e1 λx.λxs.e2)
     Cons x xs =>e2⎭
hd e                  is case e of Nil => Nil  (Cons x xs) => x
tl e                  is case e of Nil => Nil  (Cons x xs) => xs
hd                    is λz. hd z
tl                    is λz. tl z
```

Note that `case` becomes strict in the root expression `e` because it is that expression which is applied, *i.e.*, reduced, first.

Using the above syntactic shorthands, the compilation of (`Cons` e_1 e_2) to K machine code goes as follows.

$$
\begin{aligned}
&\mathcal{C}[\![\,\texttt{Cons e}_1\ \texttt{e}_2\,]\!]\rho_c\\
&\quad= \mathcal{C}[\![\,\texttt{((}\lambda\texttt{x.}\lambda\texttt{y.}\lambda\texttt{n.}\lambda\texttt{c.(c x y)) e}_1\texttt{) e}_2\,]\!]\rho_c\\
&\quad= \texttt{(App (App}\\
&\qquad\texttt{(Lam (Lam (Lam (Lam (App (App (Var 0) (Var 3)) (Var 2))))))}\\
&\qquad\mathcal{C}[\![\,\texttt{e}_1\,]\!]\rho_c\texttt{)}\ \mathcal{C}[\![\,\texttt{e}_2\,]\!]\rho_c\texttt{)}
\end{aligned}
$$

The result may look somewhat overwhelming, but the expansion is linear: there is no duplication of code. One advantage of using the encoding of data structures as functions is that when we make the K machine lazy in Section 3.5 below, the data structures will automatically be lazy too. The components of $(\texttt{Cons}\ \texttt{e}_1\ \texttt{e}_2)$ are treated as function arguments and thus are evaluated at most once.

It may seem that the compiled code for $(\texttt{Cons}\ \texttt{e}_1\ \texttt{e}_2)$ could be beta-reduced to $(\texttt{Lam}\ (\texttt{Lam}\ (\texttt{App}\ (\texttt{App}\ (\texttt{Var}\ 0)\ \texttt{C}_1)\ \texttt{C}_2)))$, where $\texttt{C}_i = \mathcal{C}[\![\, \texttt{e}_i \,]\!]\rho_c$. However, this would destroy laziness: $\texttt{e}_1$ and $\texttt{e}_2$ could be evaluated any number of times, contrary to our intentions. To see this, assume that the whnf $(\texttt{Lam}\ (\texttt{Lam}\ (\texttt{App}\ (\texttt{App}\ (\texttt{Var}\ 0)\ \texttt{C}_1)\ \texttt{C}_2)))\bullet\mathrm{E}$ of $(\texttt{Cons}\ \texttt{e}_1\ \texttt{e}_2)$ is bound to a variable `z`, and assume we are to evaluate several occurrences of the expression `(hd z)`. The expression `(hd z)` is evaluated as $(\texttt{z}\ (\lambda\texttt{n}.\lambda\texttt{c}.\texttt{n})\ (\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x}))$ which reduces to the application $(\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x})\ \texttt{e}_1\ \texttt{e}_2$, which reduces to $\texttt{e}_1$. Thus $\texttt{e}_1$ is evaluated anew at every evaluation of `(hd z)`.

3.4 Printing algebraic data structures

To make data structures really useful, we add facilities for lazy printing of data structures. This will also enable us to demonstrate a point about lazy functional languages: it is the printer which drives all computation and ultimately determines the amount and order of evaluation in a language with lazy data structures.

3.4.1 Print instructions

First, let us consider how to print the result of an expression `e` of base type. The expression will eventually reduce to its whnf which is a closure of form $\texttt{Ret}\bullet\mathrm{v}$. This whnf will activate the closure on the stack top and push itself onto the stack as described in Section 3.2.2.

By applying the base value expression `e` to the print function we achieve the desired effect. The closure activated by the base value $\texttt{Ret}\bullet\mathrm{v}$ will be the print function `PB`. The print function must now print v and activate whichever continuation is left on the stack top. Thus the print function has a side effect: it appends to the printed output. The rule for `PB` should be:

PB	E	$\texttt{Ret}\bullet\mathrm{v} : \texttt{C}_1\bullet\mathrm{E}_1 : \mathrm{S}$	$\Rightarrow$	$\texttt{C}_1$	E_1	S	and print v

Printing algebraic data structures is more complicated. Since data structures are represented as functions, type information is needed to determine how to print them, so that the user can recognize them as lists, trees, and so on in the printed output. It is impossible to distinguish at run time the encoding of the empty list `Nil` from the function $\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x}$, so it must be decided from its type whether it should print as `Nil` or as a function object. (In actual fact we do not want to print functions at all). What is more, the type must be monomorphic because a list of numbers must be printed in a different way than a list of lists of numbers.

Like a base type whnf, a data structure whnf takes functions as arguments and apply them to the components of the data structure. This means we can achieve the desired effect as above: by applying the data structure to appropriate print functions. The question is which print functions are needed and what they should do.

3.4.2 An example: Printing binary trees

Let us consider how to print a value of type `tree ::= Empty | (Fork tree tree)`. The encoding u of this value is a function (λ`e.`λ`f. ...`) of two arguments which must be functions. Applying u to print functions `PE0` and `PF0` will reduce to `PE0` if u represents `Empty`, and to (`PF0` C_1 C_2) if u represents (`Fork` C_1 C_2).

Thus the action of `PE0` must be to append "`Empty`" to the output and then activate the closure on the stack top.

The action of `PF0` must be to append "`(Fork `" to the output, then evaluate and print the first component C_1, and arrange for evaluation and printing of C_2 as well as the right parenthesis which closes the `Cons` term. To do this, `PF0` pushes three closures before activating C_1: `PE0`•[] and `PF0`•[] to print the value of C_1 (of type `tree`), and `PF1`•[] which will later take care of evaluation and printing of C_2. When C_1 has reached its whnf, it activates one of `PE0` and `PF0`. If the subtree described by C_1 is finite, `PC1`•[] will eventually be activated.

Print function `PC1` must likewise push several closures before activating C_2: `PE0` and `PF0` to take care of the printing of the whnf of C_2 (which has type `tree`) and a new one `PF2`. When C_2 has reached its whnf it activates one of `PE0` and `PF0`, and if the subtree described by C_2 is finite, `PF2` will eventually be activated, print a right parenthesis "`)`", and activate the continuation on the stack top.

In summary, the rules for these example print functions `PE0`, `PF0`, `PF1`, and `PF2` for printing `tree` values would be

PE0	E	C_1•E_1 : S	⇒	C_1	E_1	S	and print "`Empty`"
PF0	E	C_1•E_1 : S	⇒	C_1	E_1	PE0•[]:PF0•[]:PF1•[]:S	and print "`(Fork `"
PF1	E	C_1•E_1 : S	⇒	C_1	E_1	PE0•[]:PF0•[]:PF2•[]:S	and print "` `"
PF2	E	C_1•E_1 : S	⇒	C_1	E_1	S	and print "`)`"

3.4.3 Printing algebraic data structures in general

We will now describe the general scheme for an arbitrary algebraic data type. Assume we have a type definition of form

$$t \ ::= \ C_1\ t_{11}\ \dots\ t_{1c(1)} \mid \dots \mid C_n\ t_{n1}\ \dots\ t_{nc(n)}$$

We introduce $c(i)+1$ print instructions P$i0$ ... P$ic(i)$ for every constructor C_i of arity $c(i)$. The action of print instruction Pij depends on j and $c(i)$. In any case it first pops the closure on the stack top. If $j = 0 = c(i)$, it prints the constructor C_i. If $j < c(i)$ it prints a space and pushes functions to print component $j+1$ (of type $t_{i(j+1)}$). If $j = c(i) \neq 0$ it prints a right parenthesis. In any case, it activates the old stack top element. This will print the value of `Empty` as "`Empty`" and the value of `(Fork (Fork Empty Empty) Empty)` as "`(Fork (Fork Empty Empty) Empty)`".

Referring to the above example where t = `tree`, we have C_1 = `Empty` and C_2 = `Fork` with arities $c(1) = 0$ and $c(2) = 2$. The corresponding print functions are `P10` = `PE0`, `P20` = `PF0`, `P21` = `PF1`, and `P22` = `PF2`.

The print instructions for constructor C_i are defined as follows:

Pi0	E	C$_1$•E$_1$: S	⇒	C$_1$	E$_1$	prt$i$1 : Pi1•[] : S	and print "(C_i" if $c(i) \neq 0$
Pi0	E	C$_1$•E$_1$: S	⇒	C$_1$	E$_1$	S	and print "C_i" if $c(i) = 0$
Pij	E	C$_1$•E$_1$: S	⇒	C$_1$	E$_1$	prtij+1:Pij+1•[]:S	and print " " if $j < c(i)$
Pij	E	C$_1$•E$_1$: S	⇒	C$_1$	E$_1$	S	and print ")" if $j = c(i)$

Here prtij abbreviates the list of functions necessary to print a value of type t_{ij}. That is, if t_{ij} is a base type, prtij is just the closure `PB`•[], where `PB` is the K machine instruction for printing base values. If t_{ij} is an algebraic type such as t above, then prtij is a list of the functions for printing each of the possible summands: `P10`•[]:...:Pn0•[].

Note that if we disregard printing, the rules are pairwise the same (rule 1 and 3, and 2 and 4). Also, all of them activate the stack top closure.

3.4.4 Applying the general theory to printing in L

In this section we show how the general theory is adapted to printing data structures in the L language. Printing in the K machine does not follow the general theory exactly, because L is untyped.

One may identify three kinds of data in L: base values (Booleans, naturals, ...), S-expressions, and functions. A *printable value* is either a base value, or an S-expression whose components are printable values: we do not want to print functions.

printable ::= baseval | S-expr
S-expr ::= `Nil` | `Cons` printable printable

However, at run time in the K machine, S-expressions and functions are indistinguishable, so we shall assume a form of type correctness: every program evaluates to a printable value. The K machine might spuriously print function values as S-expressions, but this shall be considered a type error, and could be detected by a compile time type check.

The scheme for printing printable L values follows quite closely the printing of binary trees as discussed in Section 3.4.2 above. Only, we must design the print instructions so that they can print base values as well as S-expressions. Fortunately, base values (closures of form `Ret`•E) can be distinguished from data structures (with whnf's of form `Lam C`•E) at run time.

The `PB` (print base value) and `PN0` (print `Nil`) instructions are merged into one, called `PN`. This instruction will be activated either by a base value or by a `Nil` value. It will be able to distinguish them by looking at the stack top closure. If activated by a base value, it must remove the stack element just below the top: this will hold a `PC0` closure. If `PN` was activated by `Nil` this closure will have been removed already by `Nil`. The rules for the other instructions are those the general rules would prescribe for S-expressions.

P10	E	Ret•v : u : C₂•E₂ : S	⇒	C₂	E₂	S	and print v
P10	E	C₁•E₁ : S	⇒	C₁	E₁	S	and print “Nil”
P20	E	C₁•E₁ : S	⇒	C₁	E₁	PN•[]:PC0•[]:PC1•[]:S	and print “(Cons”
P21	E	C₁•E₁ : S	⇒	C₁	E₁	PN•[]:PC0•[]:PC2•[]:S	and print “ ”
P22	E	C₁•E₁ : S	⇒	C₁	E₁	PN•[]:PC0•[]:S	and print “)”

These rules will print the value of the L expression `(Cons (Cons 2 3) (Cons 4 Nil))` as “`(Cons (Cons 2 3) (Cons 4 Nil))`”.

3.4.5 Running the K machine with printing

In Section 3.2.3 we explained how to run the basic K machine on an expression `e`. Evaluation in the extended K machine works similarly, except that the initial stack should not be empty: it must contain print functions. Thus to evaluate `e` in initial environment E^0, the K machine is started in state (C,E^0,[P10•[], P20•[]]), where `C` is the code compiled for `e`.

If the computation terminates, the machine will stop when the print function `P10` finds the stack empty after printing a base value or “`()`”, or when `P22` finds the stack empty after printing a right parenthesis “`)`”. Notice that now the machine may never terminate and still do useful work computing and printing an infinite data structure.

3.4.6 Printing Lisp-style S-expressions

The rules just given print data structures according to the general scheme, but a more readable format à la Lisp (or Scheme) S-expressions or Miranda™ lists would be preferable. This is achieved by introducing two versions `PN` and `PN'` of instruction `P10` and two versions `PC` and `PC'` of instruction `P20`. The unprimed ones print the first component of a pair, and the primed ones print the second component. This scheme also avoids filling up the stack S with `P22` closures whose only use is to print right parentheses.

- `Nil` as a first component prints as “`()`”. As a second component it prints as “`)`”.
- A base value v as a first component prints just as v. As a second component, it prints as “ `.` v`)`” to make a dotted pair (in Lisp terminology).
- `Cons` as a first component prints as “`(`”. As a second component it prints as a blank simply: the space between two S-expressions.

PN	E	Ret•v : u : C₂•E₂ : S	⇒	C₂	E₂	S	and print v
PN	E	C₁•E₁ : S	⇒	C₁	E₁	S	and print “()”
PN'	E	Ret•v : u : C₂•E₂ : S	⇒	C₂	E₂	S	and print “ . v)”
PN'	E	C₁•E₁ : S	⇒	C₁	E₁	S	and print “)”
PC	E	C₁•E₁ : S	⇒	C₁	E₁	PN•[]:PC•[]:PCR•[]:S	and print “(”
PC'	E	C₁•E₁ : S	⇒	C₁	E₁	PN•[]:PC•[]:PCR•[]:S	and print “ ”
PCR	E	C₁•E₁:S	⇒	C₁	E₁	PN'•[]:PC'•[]:S	

These rules will print the value of the expression `(Cons (Cons 2 3) (Cons 4 Nil))` as "`((2 . 3) 4)`". This is considerably more readable than the result of the general rules above, especially for long or complicated lists.

If we disregard the printing done by these functions, `PN` and `PN'` are identical, as are `PC` and `PC'`, so the number of rules reduces to four.

To run the K machine on expression `e` with Lisp style printing, we start it in state ($\mathtt{C}$,E^0,[$\mathtt{PN}\bullet$[], $\mathtt{PC}\bullet$[]]), where `C` is the code compiled for `e`.

3.5 Adding laziness

The K machine can be made lazy in much the same way as the Three Instruction Machine. This has been previously observed by Pierre Crégut [15, p. 339]. Laziness is achieved by overwriting an argument closure $\mathtt{C}_1\bullet E_1$ (held in the environment E) by the reduced closure $\mathtt{C}_2\bullet E_2$ representing its whnf.

To achieve this overwriting we introduce *markers*, a new kind of object which can be put on the stack. Operationally speaking, a marker <p> contains a pointer p which points to a closure in an environment E. In lazy K machine, the execution of a `Var` instruction causes an argument closure $\mathtt{C}_1\bullet E_1$ to be activated, *and* pushes marker <p> onto the stack, where p points to the closure $\mathtt{C}_1\bullet E_1$ in the heap H. When the whnf $\mathtt{C}_2\bullet E_2$ of $\mathtt{C}_1\bullet E_1$ is reached and expects to take some argument from the stack, it will find the marker <p>. This causes the closure in location p to be overwritten with $\mathtt{C}_2\bullet E_2$.

The assertion that p points to $\mathtt{C}_1\bullet E_1$ in the heap is written $H[p] = \mathtt{C}_1\bullet E_1$. The action of overwriting the closure pointed to by p with another closure $\mathtt{C}_2\bullet E_2$ is written $H[p] := \mathtt{C}_2\bullet E_2$.

To formalize this in a more satisfactory manner, we should include the heap H as yet a fourth component in the K machine, and distinguish more carefully between closure construction (at execution of an application), and the passing around of pointers to environments, which are heap-allocated linked lists of heap-allocated closures. We shall not go to such a degree of formality here since we are not going to analyse or prove properties of the sharing mechanism. It is presented here mainly to indicate the intended kind of implementation.

As noted in Section 3.2.2 above, the possible whnfs have form $\mathtt{Lam}\bullet E$ or $\mathtt{Ret}\bullet v$. Both of these expect to take an argument from the stack or stop with a result, so they can conveniently test for markers before taking the argument. The new rules for the lazy K machine are shown below. The instructions not mention (`App` ...) remain as above.

`Var i`	E	S	$\Rightarrow$	$\mathtt{C}_i$	E_i	<p> : S	where $H[p] = \mathtt{C}_i\bullet E_i = E[i]$
`Lam C`	E	<p> : S	$\Rightarrow$	`Lam C`	E	S	and $H[p] :=$ `Lam C`$\bullet$E
`Lam C`	E	u:S	$\Rightarrow$	`C`	u:E	S	
`Ret`	v	<p> : S	$\Rightarrow$	`Ret`	v	S	and $H[p] :=$ `Ret`$\bullet$v
`Ret`	v	`C`$\bullet$E : S	$\Rightarrow$	`C`	E	`Ret`$\bullet$v : S	

The instructions `Lam` and `Ret`, which represent whnf's, repeatedly take a marker <p> off the stack and update the marked objects, until no markers remain. When they find a closure u on the stack top (or the stack is empty), the instructions execute as usual.

3.6 Adding letrec

We now consider the `letrec` expression, $\texttt{letrec f}_1 \texttt{ = e}_1 \ \ldots\ \texttt{f}_m \texttt{ = e}_m \texttt{ in e}_0$. A core L source expression of this form is compiled to a new K machine instruction `Lrc` as follows.

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\texttt{letrec f}_1\texttt{=e}_1\ \ldots\\ \qquad\quad \texttt{f}_m\texttt{=e}_m\ \texttt{ in e}_0\end{array}\right]\!\!\right]\rho_c \;=\; \begin{cases}\text{let } \rho_c{}' = \texttt{f}_m : \ldots : \texttt{f}_1 : \rho_c \text{ in}\\ \text{let } \texttt{C}_i \;=\mathcal{C}[\![\,\texttt{e}_i\,]\!]\rho_c{}' \text{ for } i=0,\ldots,m\\ \text{in } \texttt{Lrc (C}_1\ \ldots\ \texttt{C}_m\texttt{) C}_0\end{cases}$$

The idea in this scheme is to compile all the right hand sides $\texttt{e}_1$ through $\texttt{e}_m$ and the body expression $\texttt{e}_0$ in the same compile time environment $\rho_c{}'$, as expected for mutually recursive definitions.

The effect of the new `Lrc` instruction in environment E must be to construct a recursive run time environment E′ which extends E and provides recursive bindings for all the right hand sides. Note that the compiled right hand sides $(\texttt{C}_1\ \ldots\ \texttt{C}_m)$ are added to the run time environment E in the "natural" order, so the new environment $\mathrm{E}' = \texttt{C}_m\bullet\mathrm{E}' : \ldots : \texttt{C}_1\bullet\mathrm{E}' : \mathrm{E}$ holds them in the opposite order.

Lrc CS $\texttt{C}_0$	E	S		$\Rightarrow$	$\texttt{C}_0$	E′	S
where CS = $\texttt{C}_1\ \ldots\ \texttt{C}_m$				and whererec $\mathrm{E}' = \texttt{C}_m\bullet\mathrm{E}' : \ldots : \texttt{C}_1\bullet\mathrm{E}' : \mathrm{E}$			

It follows from the general handling of laziness with markers that a recursive data structure definition such as `letrec ones = (Cons 1 ones) in ones` will create an appropriate cyclic structure, and evaluate the expression `1` at most once.

3.7 Summary of translation and evaluation rules

Here we summarize the translation from core L into K machine code and the evaluation rules of the lazy K machine. The translation from core L to the K machine is given by:

$$\begin{array}{ll}
\mathcal{C}[\![\,\lambda\texttt{x.e}\,]\!]\rho_c & = \texttt{Lam } \mathcal{C}[\![\,\texttt{e}\,]\!](\texttt{x}{:}\rho_c)\\
\mathcal{C}[\![\,\texttt{x}_i\,]\!][\texttt{x}_0,\ldots,\texttt{x}_i,\ldots,\texttt{x}_n] & = \texttt{Var } i\\
\mathcal{C}[\![\,\texttt{e}_1\ \texttt{e}_2\,]\!]\rho_c & = \texttt{(App } \mathcal{C}[\![\,\texttt{e}_1\,]\!]\rho_c\ \mathcal{C}[\![\,\texttt{e}_2\,]\!]\rho_c\texttt{)}\\
\mathcal{C}[\![\,\texttt{c}\,]\!]\rho_c & = \texttt{Cst c}\\
\mathcal{C}[\![\,\texttt{(A e}_1\texttt{)}\,]\!]\rho_c & = \texttt{App } \mathcal{C}[\![\,\texttt{e}_1\,]\!]\rho_c\ \texttt{B1-A}\\
\mathcal{C}[\![\,\texttt{(A e}_1\ \texttt{e}_2\texttt{)}\,]\!]\rho_c & = \texttt{App(App(App } \mathcal{C}[\![\,\texttt{e}_1\,]\!]\rho_c\ \texttt{R1)}\mathcal{C}[\![\,\texttt{e}_2\,]\!]\rho_c\texttt{)B2-A}\\
\mathcal{C}[\![\,\texttt{if e}_1\ \texttt{e}_2\ \texttt{e}_3\,]\!]\rho_c & = \texttt{App(App(App } \mathcal{C}[\![\,\texttt{e}_1\,]\!]\rho_c\ \texttt{If)}\mathcal{C}[\![\,\texttt{e}_2\,]\!]\rho_c\texttt{)}\mathcal{C}[\![\,\texttt{e}_3\,]\!]\rho_c
\end{array}$$

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\texttt{letrec f}_1\texttt{=e}_1\ \ldots\\ \qquad\quad \texttt{f}_m\texttt{=e}_m\ \texttt{ in e}_0\end{array}\right]\!\!\right]\rho_c \;=\; \begin{cases}\text{let } \rho_c{}' = \texttt{f}_m : \ldots : \texttt{f}_1 : \rho_c \text{ in}\\ \text{let } \texttt{C}_i \;=\mathcal{C}[\![\,\texttt{e}_i\,]\!]\rho_c{}' \text{ for } i=0,\ldots,m \text{ in}\\ \quad \texttt{Lrc (C}_1\ \ldots\ \texttt{C}_m\texttt{) C}_0\end{cases}$$

The complete set of evaluation rules specifying the lazy K machine is:

Code C	Env. E	Stack S		Code C	Env. E	Stack S	
App C_1 C_2	E	S	$\Rightarrow$	C_1	E	$C_2\bullet E$: S	
Var i	E	S	$\Rightarrow$	C_i	E_i	<p> : S	where $H[p] = C_i\bullet E_i = E[i]$
Lam C	E	<p> : S	$\Rightarrow$	Lam C	E	S	and H[p] := Lam C$\bullet$E
Lam C	E	u:S	$\Rightarrow$	C	u:E	S	
Ret	v	<p> : S	$\Rightarrow$	Ret	v	S	and H[p] := Ret$\bullet$v
Ret	v	C$\bullet$E : S	$\Rightarrow$	C	E	Ret$\bullet$v : S	
Cst c	E	S	$\Rightarrow$	Ret	c	S	
If	E	Ret$\bullet$T:$C_1\bullet E_1$:$C_2\bullet E_2$:S	$\Rightarrow$	C_1	E_1	S	
If	E	Ret$\bullet$F:$C_1\bullet E_1$:$C_2\bullet E_2$:S	$\Rightarrow$	C_2	E_2	S	
B1-op	E'	Ret$\bullet$v : S	$\Rightarrow$	Ret	(op v)	S	
R1	E'	Ret$\bullet v_1$: C$\bullet$E : u : S	$\Rightarrow$	C	E	u : Ret$\bullet v_1$: S	
B2-op	E'	Ret$\bullet v_2$: Ret$\bullet v_1$: S	$\Rightarrow$	Ret	(op v_1 v_2)	S	
PN	E	Ret$\bullet$v : u : $C_2\bullet E_2$: S	$\Rightarrow$	C_2	E_2	S	and print v
PN	E	$C_1\bullet E_1$: S	$\Rightarrow$	C_1	E_1	S	and print "()"
PN'	E	Ret$\bullet$v : u : $C_2\bullet E_2$: S	$\Rightarrow$	C_2	E_2	S	and print " . v)"
PN'	E	$C_1\bullet E_1$: S	$\Rightarrow$	C_1	E_1	S	and print ")"
PC	E	$C_1\bullet E_1$: S	$\Rightarrow$	C_1	E_1	PN$\bullet$[]:PC$\bullet$[]:PCR$\bullet$[]:S	and print "("
PC'	E	$C_1\bullet E_1$: S	$\Rightarrow$	C_1	E_1	PN$\bullet$[]:PC$\bullet$[]:PCR$\bullet$[]:S	and print " "
PCR	E	$C_1\bullet E_1$: S	$\Rightarrow$	C_1	E_1	PN'$\bullet$[]:PC'$\bullet$[]:S	
Lrc CS C_0	E	S	$\Rightarrow$	C_0	E'	S	
where CS = C_1 ... C_m			and wherere$c$ $E' = C_m\bullet E'$: ... : $C_1\bullet E'$: E				

We have written an experimental implementation of the compilation from L to K machine code, and of the K machine itself. The implementation and its use are briefly described in Appendix B.1.

Chapter 4

Closure Analysis

This chapter describes an approximate program analysis called *closure analysis*. The purpose of the closure analysis is to find a superset of the set of closures (or functions) to which a given expression may evaluate. This information is given as a set of the textual lambdas in the program which may generate such closures. The closure analysis is defined on the lazy language L introduced in Chapter 3, but works both for strict and lazy languages.

The closure analysis is useful for finding an approximation to the set of arguments to which a given lambda may be applied. It is used in the usage count analysis (Chapter 5), and in the globalization algorithm (Chapter 7). It is useful in general to extend analyses for first order languages to work also for higher order languages, as explained in Section 4.4 below.

The analysis originates in my Master's thesis [58] where is was proved correct for a strict language. Variants of it have been used in other work also [10] [40] [67] [31].

The version presented here has been extended to work for data structures, lambdas, and letrec, and is proved correct with respect to the K machine in Section 4.2.

A similar analysis for the language Scheme has been developed independently by Shivers [60] [63].

4.1 Analysis of the core L language

We shall show how to do closure analysis for the core L language. First we describe the analysis for the language except `letrec`, then we describe the (straightforward) extension to handle it in Section 4.1.2 below. Data structures are covered by the standard translation into the core language and require no special treatment. It is interesting to observe, however, that this simple-minded approach gives quite reasonable results (Section 4.1.3).

4.1.1 Closure analysis

For the closure analysis we shall require that all lambdas λ`x.e` in the program are uniquely labelled: λ^ℓ`x.e`. Furthermore, every variable occurrence `x` in the program is labelled with the label ℓ of the lambda binding the variable: $\mathtt{x}^\ell$. In the discussion below we shall sometimes identify lambdas and variables with their labels.

Semantically speaking, the label ℓ of a lambda expression $\lambda^\ell\mathtt{x.e}$ abstracts the set of all closures `Lam C`•E that can be formed from the lambda expression by choosing an arbitrary environment E, that is, choosing arbitrary bindings for the free variables of the lambda expression.

We shall assume that no closure `C`•E in the initial environment E^0 of a computation can evaluate to a function. A desirable consequence is that there can be no labels in the run time input, so the set of labels necessary can be determined from the program text alone. A negative consequence is that function values and hence data structure values cannot be given as input to a program.

Let a fixed L program $\mathtt{e}_p$ be given, and let Label be the set of labels. In the analysis we use two *closure descriptions* ϕ, ρ: CDescription = Label $\rightarrow \wp$(Label). These are called the *result closure description* (ϕ) and the *argument closure description* (ρ) and are intended to give the following information:

$$\begin{array}{lcl} \phi\ \ell & = & \text{the set of lambdas that the body } \mathtt{e}_1 \text{ of } \lambda^\ell\mathtt{x.e}_1 \text{ can evaluate to} \\ \rho\ \ell & = & \text{the set of lambdas that } \lambda^\ell\mathtt{x.e}_1 \text{ can be applied to} \\ & = & \text{the set of lambdas that variable } \mathtt{x}^\ell \text{ can be bound to} \end{array}$$

We shall define two analysis functions, $\mathcal{P}_e$ and $\mathcal{P}_v$, with the following meanings:

$$\begin{array}{lcl} \mathcal{P}_e[\![\,\mathtt{e}\,]\!]\phi\rho & = & \text{the set of lambdas that expression } \mathtt{e} \text{ can evaluate to} \\ \mathcal{P}_v[\![\,\mathtt{e}\,]\!]\phi\rho\ell' & = & \text{the set of lambdas that lambda } \ell' \text{ can be applied to in } \mathtt{e} \end{array}$$

The $\mathcal{P}_e$ function is called the *closure analysis function* and the $\mathcal{P}_v$ is called the *closure propagation function.* The closure analysis function $\mathcal{P}_e$ is defined as follows:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathcal{P}_e : \text{Expression} \rightarrow \text{CDescription} \rightarrow \text{CDescription} \rightarrow \wp(\text{Label})} \\ \mathcal{P}_e[\![\,\lambda^\ell\mathtt{x.e}\,]\!]\phi\rho & = & \{\ \ell\ \} \\ \mathcal{P}_e[\![\,\mathtt{x}^\ell\,]\!]\phi\rho & = & \rho\ \ell \\ \mathcal{P}_e[\![\,\mathtt{(e_1\ e_2)}\,]\!]\phi\rho & = & \bigcup\{\ \phi\ \ell \mid \ell \in \mathcal{P}_e[\![\,\mathtt{e}_1\,]\!]\phi\rho\ \} \\ \mathcal{P}_e[\![\,\mathtt{(A\ e_1\ \ldots\ e}_n\mathtt{)}\,]\!]\phi\rho & = & \{\} \\ \mathcal{P}_e[\![\,\mathtt{if\ e_1\ e_2\ e_3}\,]\!]\phi\rho & = & \mathcal{P}_e[\![\,\mathtt{e}_2\,]\!]\phi\rho \cup \mathcal{P}_e[\![\,\mathtt{e}_3\,]\!]\phi\rho \end{array}$$

These analysis equations are justified as follows. The lambda expression labelled ℓ can evaluate only to itself. A variable labelled ℓ can evaluate to every closure the variable can be bound to, that is, ($\rho\ \ell$). The application $\mathtt{(e_1\ e_2)}$ can evaluate to those closures which an application of lambda ℓ can, where ℓ is a possible closure value of $\mathtt{e}_1$. A fully applied base function must return a base value and so cannot evaluate to a lambda. A conditional can evaluate only to those lambdas to which any of the two branches can evaluate.

The closure propagation function $\mathcal{P}_v$ is defined as follows:

$$
\begin{array}{lcl}
\mathcal{P}_v : \text{Expression} \to \text{CDescription} \to \text{CDescription} \to \text{Label} \to \wp(\text{Label}) & & \\
\mathcal{P}_v[\![\, \lambda^{\ell}\texttt{x.e} \,]\!]\phi\rho\ell' & = & \mathcal{P}_v[\![\, \texttt{e} \,]\!]\phi\rho\ell' \\
\mathcal{P}_v[\![\, \texttt{x}^{\ell} \,]\!]\phi\rho\ell' & = & \{\} \\
\mathcal{P}_v[\![\, \texttt{(e}_1\ \texttt{e}_2\texttt{)} \,]\!]\phi\rho\ell' & = & \mathcal{P}_v[\![\, \texttt{e}_1 \,]\!]\phi\rho\ell' \cup \mathcal{P}_v[\![\, \texttt{e}_2 \,]\!]\phi\rho\ell' \\
 & & \cup \bigcup \{\ \mathcal{P}_e[\![\, \texttt{e}_2 \,]\!]\phi\rho \mid \ell' \in \mathcal{P}_e[\![\, \texttt{e}_1 \,]\!]\phi\rho\ \} \\
\mathcal{P}_v[\![\, \texttt{(A e}_1\ \ldots\ \texttt{e}_n\texttt{)} \,]\!]\phi\rho\ell' & = & \bigcup_{j=1}^{n} \mathcal{P}_v[\![\, \texttt{e}_j \,]\!]\phi\rho\ell' \\
\mathcal{P}_v[\![\, \texttt{if e}_1\ \texttt{e}_2\ \texttt{e}_3 \,]\!]\phi\rho\ell' & = & \bigcup_{j=1}^{3} \mathcal{P}_v[\![\, \texttt{e}_j \,]\!]\phi\rho\ell'
\end{array}
$$

These equations are justified as follows: The closures that lambda ℓ' can be applied to in a lambda expression are those it can be applied to in its body. In a variable expression, it cannot be applied. In an application $\texttt{(e}_1\ \texttt{e}_2\texttt{)}$, ℓ' can be applied to whatever it can be applied to in $\texttt{e}_1$ or $\texttt{e}_2$. In addition there are two possibilities for the contribution from the application itself: Either $\texttt{e}_1$ can evaluate to ℓ', in which case ℓ' can be applied to whatever $\texttt{e}_2$ can evaluate to. Or $\texttt{e}_1$ cannot evaluate to ℓ', in which case there is no contribution from the application itself. In a base function application, ℓ' can be applied to whatever it can be applied to in any of the argument expressions. In a conditional expression, ℓ' can be applied to whatever it can be applied to in any of the three component expressions.

To analyse a program `e` we seek descriptions ϕ and ρ which are safe and not unnecessarily conservative. Description ϕ_1 is no more conservative than ϕ_2, written $\phi_1 \sqsubseteq \phi_2$, if and only if for all labels $\ell \in$ Label, $\phi_1\ \ell \subseteq \phi_2\ \ell$. The same definition applies to ρ.

Thus we seek the descriptions ϕ and ρ which constitute the $\sqsubseteq$-least simultaneous solution to the equations

$$
\begin{array}{lcll}
\phi\ \ell & = & \mathcal{P}_e[\![\, \texttt{e}_1 \,]\!]\phi\rho & \text{for all lambdas } \lambda^{\ell}\texttt{x.e}_1 \text{ in } \texttt{e} \\
\rho\ \ell & = & \mathcal{P}_v[\![\, \texttt{e} \,]\!]\phi\rho\ell & \text{for all lambdas } \lambda^{\ell}\texttt{x.e}_1 \text{ in } \texttt{e}
\end{array}
$$

4.1.2 Closure analysis for letrec

Local recursive (`letrec`) definitions in the core language introduce new names in the same manner as lambda expressions. Thus to analyse the expression $\texttt{letrec f}_1\ \texttt{=}\ \texttt{e}_1\ \ldots\ \texttt{f}_m\ \texttt{=}\ \texttt{e}_m\ \texttt{in e}_0$, we require that also the `letrec` bindings are labelled uniquely with labels $\ell \in$ Label. The set of `letrec` labels is disjoint from the set of lambda labels. For such a labelled recursive binding $\texttt{f} =^{\ell} \texttt{e}$, the meaning of ϕ and ρ is:

$$
\begin{array}{lcl}
\phi\ \ell & = & \text{the set of lambdas that } \texttt{e} \text{ can evaluate to} \\
 & = & \text{the set of lambdas that } \texttt{f} \text{ can be bound to} \\
 & = & \rho\ \ell
\end{array}
$$

This can be understood by considering the explanation of recursive bindings as lambda expressions, as follows. The value of `e` as defined by the recursive binding $\texttt{letrec f} =^{\ell} \texttt{e in} \ldots$ is $\texttt{(Y}\ \lambda^{\ell}\texttt{f.e)}$ which satisfies $\texttt{((}\lambda^{\ell}\texttt{f.e)(Y}\ \lambda^{\ell}\texttt{f.e))} = \texttt{(Y}\ \lambda^{\ell}\texttt{f.e)}$. From this equation two things are clear. First, by reading the equation forwards we see that a possible result of applying $\lambda^{\ell}\texttt{f.e}$ is $\texttt{(Y}\ \lambda^{\ell}\texttt{f.e)}$. That is, $\phi\ \ell$ must contain all the lambdas that `e` can evaluate to. Secondly, by considering the application of $\lambda^{\ell}\texttt{f.e}$, we see that a possible argument and hence a possible value of `f` is `(Y` λ^ℓ`f.e)`. That is, $\rho\ \ell$ must contain all the lambdas that `e` can evaluate to.

Thus to analyse a program containing `letrec`, we seek the descriptions ϕ and ρ which constitute the $\sqsubseteq$-least simultaneous solution to the equations

$$\begin{array}{lll} \phi\ \ell & = \mathcal{P}_e[\![\, \mathtt{e}_1 \,]\!]\phi\rho & \text{for all lambdas } \lambda^\ell\mathtt{x.e}_1 \text{ in } \mathtt{e} \\ \rho\ \ell & = \mathcal{P}_v[\![\, \mathtt{e} \,]\!]\phi\rho\ell & \text{for all lambdas } \lambda^\ell\mathtt{x.e}_1 \text{ in } \mathtt{e} \\ \phi\ \ell_i & = \rho\ \ell_i = \mathcal{P}_e[\![\, \mathtt{e}_i \,]\!]\phi\rho & \text{for all } \mathtt{letrec}\ \mathtt{f}_1\mathtt{=}^{\ell_1}\mathtt{e}_1\ \ldots\ \mathtt{f}_m\mathtt{=}^{\ell_m}\mathtt{e}_m\ \mathtt{in}\ \mathtt{e}_0 \\ & & \text{and for all } i = 1, \ldots, m \end{array}$$

The additional equations for the analysis functions $\mathcal{P}_e$ and $\mathcal{P}_v$ are

$$\begin{array}{ll} \mathcal{P}_e[\![\, \mathtt{letrec}\ \mathtt{f}_1\mathtt{=}^{\ell_1}\mathtt{e}_1\ldots\mathtt{f}_m\mathtt{=}^{\ell_m}\mathtt{e}_m\ \mathtt{in}\ \mathtt{e}_0 \,]\!]\phi\rho & = \mathcal{P}_v[\![\, \mathtt{e}_0 \,]\!]\phi\rho \\ \mathcal{P}_v[\![\, \mathtt{letrec}\ \mathtt{f}_1\mathtt{=}^{\ell_1}\mathtt{e}_1\ldots\mathtt{f}_m\mathtt{=}^{\ell_m}\mathtt{e}_m\ \mathtt{in}\ \mathtt{e}_0 \,]\!]\phi\rho\ell' & = \bigcup_{i=0}^{m} \mathcal{P}_v[\![\, \mathtt{e}_i \,]\!]\phi\rho\ell' \end{array}$$

4.1.3 The handling of data structures

The restriction that the run time input (the initial environment) cannot contain functions is rather standard. Because of our encoding of data structures as functions, it also prevents data structures from being input to a program. Obviously, this restricts the usefulness of the analysis somewhat, but we have found no easy way around this.

The essence of the problem is that data structure input would have to be labelled. This requirement would not agree with the view that the user can consider an automatic program analysis as a black box, the behaviour and workings of which he need not understand.

To see that the analysis yields reasonable results on programs using a combination of higher order functions and data structures, consider the closure analysis of the following program. It produces an infinite list of numbers by applying an infinite list `(multiples 100)` of functions to the argument 3. (The superscripts 7 and 19 are labels).

```
letrec multiples = λ n.(Cons (λ⁷x.(* x n)) (multiples (+ n 1)))
       map       = λ g.λ l.case l of
                               Nil        => Nil
                               (Cons x r) => Cons (g x) (map g r)
in     map (λ¹⁹f.(f 3)) (multiples 100)
```

Closure analysis of this program tells us that `g` in `map` can be bound (only) to $(\lambda^{19}$`f.(f 3))`, that the (only) function `x` in `map` can bind to is $(\lambda^{7}$`x.(* x n))`, and hence that the (only) function value of `f` is $(\lambda^{7}$`x.(* x n))`. This is satisfactory, given that this is not evident from the syntax of the program.

4.2 Correctness with respect to the K machine

In this section we shall prove the closure analysis correct with respect to the K machine. The analysis cannot be proven correct with respect to a standard denotational semantics, because the standard denotational semantics would treat functions extensionally: it does not matter *how* a function is specified; only its extension, roughly the set of (argument,value) pairs, matters. This means that from the standard semantics value of an expression it is impossible to find which lambdas may generate its value.

Using an operational semantics or abstract machine is better for this purpose, as the description of a function value may, and usually will, incorporate information about which textual lambda generated it. Thus we shall assume that K machine computations carry around labelled lambdas $\mathtt{Lam}^{\ell}\ \mathtt{C}$; this is needed in the correctness proof given below.

The original closure analysis for a call by value language was proved correct in [58] with respect to another operational semantics, a so-called natural semantics [43] for a strict language.

An improvement is that the present correctness proof is valid for non-terminating computations, which the original one was not. In a side-effect free strict language (without lazy data structures) non-terminating computations do not make sense. We therefore usually identify all non-terminating computations with the "undefined" or (pragmatically speaking) "useless" result. However, in a language with lazy data structures it is possible to distinguish those non-terminating computations which compute an infinite data structure. This is because lazy printing allows us to look at arbitrary finite prefixes of (*i.e.*, approximations to) the infinite result.

4.2.1 Correctness of closure analysis

The closure analysis is correct if it safely describes the set of lambdas to which an expression can evaluate. Thus if expression e can evaluate to $\lambda^{\ell}\mathtt{x.e}$, then the closure analysis function must satisfy $\ell \in \mathcal{P}_e[\![\mathtt{e}]\!]\phi\rho$.

The plan of the proof is as follows. We start by proving the analysis correct in detail for the applicative kernel: the `Lam`, `Var`, `App` subset of the K machine instructions. Then we explain the modifications needed for base values and `letrec`.

First we must define a number of auxiliary concepts: Trace of the K machine, computation, whnf computation, and balanced trace. We then prove a property of balanced subtraces by induction on the length of a trace. This property is then used to prove the correctness of the analysis: if e can evaluate to $\lambda^{\ell}\mathtt{x.e}$, then $\ell \in \mathcal{P}_e[\![\mathtt{e}]\!]\phi\rho$.

4.2.2 Preliminaries

A *trace* of the K machine is a finite or infinite sequence $(\mathtt{C}^0,\mathrm{E}^0,\mathrm{S}^0)\ \ldots\ (\mathtt{C}^n,\mathrm{E}^n,\mathrm{S}^n)\ \ldots$ of K machine states, such that any two succeeding states are related by a rule of the K machine.

A *subtrace* of a trace is a contiguous subsequence. An *n-prefix* is an initial subtrace $(\mathtt{C}^0,\mathrm{E}^0,\mathrm{S}^0)\ \ldots\ (\mathtt{C}^n,\mathrm{E}^n,\mathrm{S}^n)$, *i.e.*, one that has n state transitions and $n+1$ states.

A *computation* of the K machine for a given L program e is a trace $(\mathtt{C}^0,\mathrm{E}^0,\mathrm{S}^0)\ \ldots\ (\mathtt{C}^n,\mathrm{E}^n,\mathrm{S}^n)\ \ldots$ for which $\mathtt{C}^0 = \mathcal{C}[\![\mathtt{e}]\!]\rho_c$ is the compiled code for e, E^0 is the initial environment, and S^0=[] is the empty stack. Recall that run time input to a program is given through the free variables of e; the initial environment E^0 provides values for these variables.

A *balanced trace* is a finite trace ($\mathtt{C}^0$,E^0,S^0) ... ($\mathtt{C}^n$,E^n,S^n) for which the sequence $\mathtt{C}^0$ ... $\mathtt{C}^{n-1}$ of the K machine instructions has the form baltrace where

baltrace ::= simple | baltrace baltrace
simple ::= `Var` | `App` baltrace `Lam`

A balanced trace is *simple* if the sequence of instructions has form simple; otherwise it is *composite*. In a simple balanced trace ($\mathtt{C}^0$,E^0,S^0) ... ($\mathtt{C}^n$,E^n,S^n), the initial and the final stacks are identical: $S^0 = S^n$. This fact is easily proven by induction on the "compositeness" of a balanced trace and by considering the K machine rules for `Var`, `App`, and `Lam`. This means that the initial and final instructions $\mathtt{C}^0$ and $\mathtt{C}^n$ have the same continuation, or are in the same context. A balanced subtrace represents a subcomputation such as a step in the evaluation of an expression towards its whnf.

A *whnf computation* for instruction $\mathtt{C}^0$ is the shortest balanced trace ($\mathtt{C}^0$,E^0,S^0) ... ($\mathtt{C}^n$,E^n,S^n) for which $\mathtt{C}^n$ has form $\mathtt{Lam}^\ell\ \mathtt{C}'$. In this case, the whnf of $\mathtt{C}^0\bullet E^0$ is $\mathtt{Lam}^\ell\ \mathtt{C}'\bullet E^n$. Note that $S^n = S^0$ also in this case, so the contexts of the two closures are the same.

A small notational problem is that the analysis functions work on labelled L language expressions whereas the machine executes K machine instructions. We use a labelled translation to transfer labels from L expressions to K machine instructions:

$$
\begin{array}{lcl}
\mathcal{C}[\![\lambda^\ell \mathtt{x.e}]\!]\rho_c & = & \mathtt{Lam}^\ell\ \mathcal{C}[\![\mathtt{e}]\!](\mathtt{x}:\rho_c) \\
\mathcal{C}[\![\mathtt{x}^\ell{}_i]\!][\mathtt{x}_0,\ldots,\mathtt{x}_i,\ldots,\mathtt{x}_n] & = & \mathtt{Var}^\ell\ i \\
\mathcal{C}[\![\mathtt{e}_1\ \mathtt{e}_2]\!]\rho_c & = & (\mathtt{App}\ \mathcal{C}[\![\mathtt{e}_1]\!]\rho_c\ \mathcal{C}[\![\mathtt{e}_2]\!]\rho_c)
\end{array}
$$

Below we shall occasionally be sloppy and apply the analysis functions to the compiled instructions also. We may say *e.g.* "$\mathcal{P}_e[\![\mathtt{Lam}^\ell\ \mathtt{C}]\!]$" when we mean "$\mathcal{P}_e[\![\lambda^\ell \mathtt{x.e}]\!]$" where $\mathtt{C} \equiv \mathcal{C}[\![\mathtt{e}]\!]\rho_c$ is the code compiled for `e`, and ρ_c is a suitable compile time environment.

4.2.3 Correctness theorem for the applicative kernel

Let a fixed program $\mathtt{e}_p$ in the applicative kernel of L be given, with corresponding K machine code $\mathtt{C}_p$. We shall prove the closure analysis correct for this program. Let therefore ϕ and ρ be the descriptions computed for $\mathtt{e}_p$. We start with the following

Lemma 1 For every balanced subtrace ($\mathtt{C}$,E,S) ... ($\mathtt{C}'$,E$'$,S) of a computation for $\mathtt{e}_p$ it holds that $\mathcal{P}_e[\![\mathtt{C}]\!]\phi\rho \supseteq \mathcal{P}_e[\![\mathtt{C}']\!]\phi\rho$.

Proof See Section 4.2.4 below. □

From this lemma the desired correctness of the closure analysis follows:

Proposition 1 Let ($\mathtt{C}$,E,S) ... ($\mathtt{Lam}^{\ell'}\ \mathtt{C}'$,E$'$,S) be a whnf computation for $\mathtt{C} = \mathcal{C}[\![\mathtt{e}]\!]$, where `e` is a subexpression of $\mathtt{e}_p$. Then $\ell' \in \mathcal{P}_e[\![\mathtt{e}]\!]\phi\rho$.

Proof By the above lemma, $\mathcal{P}_e[\![\, \mathtt{e} \,]\!]\phi\rho = \mathcal{P}_e[\![\, \mathtt{C} \,]\!]\phi\rho \supseteq \mathcal{P}_e[\![\, \mathtt{Lam}^{\ell'}\ \mathtt{C'} \,]\!]\phi\rho = \{\ \ell'\ \}$. □

That is, the closure analysis function $\mathcal{P}_e$ computes a superset of the set of lambdas to which `e` may evaluate; it safely approximates that set. The result about balanced traces is proved for arbitrary computations from $\mathtt{C}_p$ by induction on their n-prefixes. To state the induction hypothesis, we need two more auxiliary concepts.

An environment E is *ρ-safe* for the analysis if

- for every variable $\mathtt{x}^\ell$ which has an associated value $\mathtt{C}_i \bullet \mathrm{E}_i$ in $\mathrm{E} = \mathtt{C}_1 \bullet \mathrm{E}_1 : \ldots : \mathtt{C}_n \bullet \mathrm{E}_n$, $\rho\ \ell \supseteq \mathcal{P}_e[\![\, \mathtt{C}_i \,]\!]\phi\rho$, and
- every E_i is ρ-safe, for $i = 1, \ldots, n$.

A stack $\mathrm{S} = \mathtt{C}_1 \bullet \mathrm{E}_1 : \ldots : \mathtt{C}_n \bullet \mathrm{E}_n$ is *E-safe* if every E_i is ρ-safe, for $i = 1, \ldots, n$.

4.2.4 Proof of the Lemma

The induction hypothesis for an n-prefix $(\mathtt{C}^0,\mathrm{E}^0,\mathrm{S}^0) \ldots (\mathtt{C}^n,\mathrm{E}^n,\mathrm{S}^n)$ has two parts.

(1) Every simple balanced subtrace $(\mathtt{C},\mathrm{E},\mathrm{S}) \ldots (\mathtt{C'},\mathrm{E'},\mathrm{S})$ of the n-prefix satisfies $\mathcal{P}_e[\![\, \mathtt{C} \,]\!]\phi\rho \supseteq \mathcal{P}_e[\![\, \mathtt{C'} \,]\!]\phi\rho$.

(2) In every state $(\mathtt{C}^i,\mathrm{E}^i,\mathrm{S}^i)$ of the n-prefix, E^i is ρ-safe and S^i is E-safe.

From the property $\mathcal{P}_e[\![\, \mathtt{C} \,]\!]\phi\rho \supseteq \mathcal{P}_e[\![\, \mathtt{C'} \,]\!]\phi\rho$ for simple balanced subtraces, the same property follows for composite ones. This is proved by considering the decomposition into simple balanced subtraces and using transitivity of $\supseteq$. This will be exploited below.

Base case, $n = 0$: The 0-prefix is just $(\mathtt{C}_p,\mathrm{E}^0,[])$. This contains no balanced subtrace, so (1) holds trivially. From the assumption that E^0 contains no function values, we have $\mathcal{P}_e[\![\, \mathtt{C}_i \,]\!]\phi\rho = \{\}$ for all $\mathtt{C}_i$ in E^0; hence (2) holds.

Induction step, $n > 0$: Assuming the induction hypothesis for the n-prefix, we show it for the $(n+1)$-prefix by case analysis on the instruction $\mathtt{C}^n$ in the last state of the n-prefix.

Case $\mathtt{C}^n \equiv \mathtt{Var}^\ell\ \mathtt{i}$: The prefix is extended with the transition $(\mathtt{Var}^\ell\ \mathtt{i},\mathrm{E},\mathrm{S}) \Rightarrow (\mathtt{C}_i,\mathrm{E}_i,\mathrm{S})$. The only new simple balanced subtrace is $(\mathtt{Var}^\ell\ \mathtt{i},\mathrm{E},\mathrm{S}) \Rightarrow (\mathtt{C}_i,\mathrm{E}_i,\mathrm{S})$, but $\mathcal{P}_e[\![\, \mathtt{Var}^\ell\ \mathtt{i} \,]\!]\phi\rho = \rho\ \ell \supseteq \mathcal{P}_e[\![\, \mathtt{C}_i \,]\!]\phi\rho$, by definition of $\mathcal{P}_e$ on variables, and by ρ-safety of E. From this it follows that property (1) holds also for the $(n+1)$-prefix.

From ρ-safety of E follows the ρ-safety of E_i, hence property (2) holds.

Case $\mathtt{C}^n \equiv \mathtt{Lam}^{\ell'}\ \mathtt{C'}$: If the stack S^n is empty, then the trace cannot be extended, so we assume it is non-empty. The prefix is extended with the transition $(\mathtt{Lam}^{\ell'}\ \mathtt{C'},\mathrm{E},\mathtt{C}_2 \bullet \mathrm{E}_2{:}\mathrm{S}) \Rightarrow (\mathtt{C'},\mathtt{C}_2 \bullet \mathrm{E}_2{:}\mathrm{E},\mathrm{S})$. The extended prefix contains one new simple balanced subtrace, for which the sequence of instructions $\mathtt{C}^i$ has form `(App C`$_1$

$\texttt{C}_2$) $\texttt{C}_1 \ldots$ ($\texttt{Lam}^{\ell'}$ $\texttt{C}'$) $\texttt{C}'$ where the subtrace $\texttt{C}_1 \ldots$ ($\texttt{Lam}^{\ell'}$ $\texttt{C}'$) is itself a (possibly composite) balanced trace. Hence by part (1) of the induction hypothesis (and the note about composite traces above), $\mathcal{P}_e[\![\,\texttt{C}_1\,]\!]\phi\rho \supseteq \mathcal{P}_e[\![\,\texttt{Lam}^{\ell'}\ \texttt{C}'\,]\!]\phi\rho$, so $\ell' \in \mathcal{P}_e[\![\,\texttt{C}_1\,]\!]\phi\rho$ by definition of $\mathcal{P}_e$ on lambdas.

To see that property (1) holds, we must show $\mathcal{P}_e[\![\,\texttt{App C}_1\ \texttt{C}_2\,]\!]\phi\rho \supseteq \mathcal{P}_e[\![\,\texttt{C}'\,]\!]\phi\rho$. But $\mathcal{P}_e[\![\,\texttt{App C}_1\ \texttt{C}_2\,]\!]\phi\rho = \bigcup\{\ \phi\ \ell \mid \ell \in \mathcal{P}_e[\![\,\texttt{C}_1\,]\!]\phi\rho\ \} \supseteq \phi\ \ell' = \mathcal{P}_e[\![\,\texttt{C}'\,]\!]\phi\rho$, by definition of $\mathcal{P}_e$ on applications, by $\ell' \in \mathcal{P}_e[\![\,\texttt{C}_1\,]\!]\phi\rho$, and by definition of ϕ.

To see that property (2) holds we must prove that $\rho\ \ell' \supseteq \mathcal{P}_e[\![\,\texttt{C}_2\,]\!]\phi\rho$; then the ρ-safety of $\texttt{C}_2\bullet\text{E}_2$:E will follow from this, the ρ-safety of E, and the E-safety of S^n = $\texttt{C}_2\bullet\text{E}_2$:S. But $\rho\ \ell' = \mathcal{P}_v[\![\,\texttt{e}_p\,]\!]\phi\rho\ell' \supseteq \mathcal{P}_v[\![\,\texttt{App C}_1\ \texttt{C}_2\,]\!]\phi\rho\ell' = \bigcup\ \{\ \mathcal{P}_e[\![\,\texttt{C}_2\,]\!]\phi\rho \mid \ell' \in \mathcal{P}_e[\![\,\texttt{C}_1\,]\!]\phi\rho\ \} = \mathcal{P}_e[\![\,\texttt{C}_2\,]\!]\phi\rho$, by definition of ρ, by properties of $\mathcal{P}_v$, by definition of $\mathcal{P}_v$ on applications, and by $\ell' \in \mathcal{P}_e[\![\,\texttt{C}_1\,]\!]\phi\rho$.

Case $\texttt{C}^n \equiv \texttt{App C}_1\ \texttt{C}_2$: The prefix is extended by the transition ($\texttt{App C}_1\ \texttt{C}_2$,E,S) $\Rightarrow$ ($\texttt{C}_1$,E,$\texttt{C}_2\bullet$E:S). Property (1) holds because no new balanced traces are introduced. Property (2) holds because the E-safety of $\texttt{C}_2\bullet$E:S follows from the ρ-safety of E_2 and the E-safety of S.

This completes the proof by cases of the inductive step.

This completes the induction proof of Lemma 1. Hence property (1) of balanced traces holds for arbitrary computations, even infinite ones. The correctness of the closure analysis function $\mathcal{P}_e$ for the applicative kernel of L follows as noted in Proposition 1 above. The subsections below argue that the analysis is correct also for the rest of core L: the constructs for handling base values, and `letrec`.

4.2.5 Correctness for base value instructions

The correctness of the closure analysis for those parts of core L dealing with base values is demonstrated in this section. We shall not give proof in full detail as above. Instead we show how to adapt the notions of balanced trace and whnf computation introduced above to the extended language and the extended K machine.

4.2.5.1 Preliminaries revisited

The concepts of balanced trace and whnf computation need some revision to take into account the additional L expressions and K machine instructions.

There are a few additional forms of simple balanced traces, corresponding to evaluation of base value constants, one- and two-argument base function applications, and conditional.

simple ::= `Var` | `App` baltrace `Lam`
| `Cst`
| `App` baltrace `Ret B1-A`
| `App App App` baltrace `Ret R1` baltrace `Ret B2-A`
| `App App App` baltrace `Ret If`

It is not hard to see that the four new forms of simple balanced traces correspond precisely to the trace of evaluating the four constructs in core L which deal with base values:

$$
\begin{array}{ll}
\mathcal{C}[\![\,\texttt{c}\,]\!]\rho_c & = \texttt{Cst c} \\
\mathcal{C}[\![\,\texttt{(A e}_1\texttt{)}\,]\!]\rho_c & = \texttt{App}\ \mathcal{C}[\![\,\texttt{e}_1\,]\!]\rho_c\ \texttt{B1-A} \\
\mathcal{C}[\![\,\texttt{(A e}_1\ \texttt{e}_2\texttt{)}\,]\!]\rho_c & = \texttt{App (App (App}\ \mathcal{C}[\![\,\texttt{e}_1\,]\!]\rho_c\ \texttt{R1)}\ \mathcal{C}[\![\,\texttt{e}_2\,]\!]\rho_c\texttt{)}\ \texttt{B2-A} \\
\mathcal{C}[\![\,\texttt{if e}_1\ \texttt{e}_2\ \texttt{e}_3\,]\!]\rho_c & = \texttt{(App (App (App}\ \mathcal{C}[\![\,\texttt{e}_1\,]\!]\rho_c\ \texttt{If)}\ \mathcal{C}[\![\,\texttt{e}_2\,]\!]\rho_c\texttt{)}\ \mathcal{C}[\![\,\texttt{e}_3\,]\!]\rho_c\texttt{)}
\end{array}
$$

We shall assume that expressions involving base values and base functions are type correct so that the traces for evaluation of base value expressions must end with a `Ret` instruction, not `Lam`. Using this assumption it is easy to prove that the evaluation stack is preserved across a balanced trace $(\texttt{C}^0,\mathrm{E}^0,\mathrm{S}^0)\ \ldots\ (\texttt{C}^n,\mathrm{E}^n,\mathrm{S}^n)$: it holds that $\mathrm{S}^0 = \mathrm{S}^n$.

The introduction of a new kind of whnf besides those of form `Lam` necessitates a revision of the definition of whnf computation. A *whnf computation* for $\texttt{C}^0$ is the shortest balanced trace $(\texttt{C}^0,\mathrm{E}^0,\mathrm{S}^0)\ \ldots\ (\texttt{C}^n,\mathrm{E}^n,\mathrm{S}^n)$ for which $\texttt{C}^n$ has form $\texttt{Lam}^\ell\texttt{C}'$ or `Ret`. In the former case the whnf of $\texttt{C}^0\bullet\mathrm{E}^0$ is the function $\texttt{Lam}^\ell\texttt{C}'\bullet\mathrm{E}^n$, and in the latter case it is the base value $\texttt{Ret}\bullet\mathrm{E}^n$.

4.2.5.2 Extension to the proof of the Lemma

We extend the case analysis in the proof given in Section 4.2.4 above.

Case $\texttt{C}^n \equiv$ `Ret` or `R1`: This does not introduce any new simple balanced traces, hence property (1) is preserved. The swapping of environments and stack top elements does not affect property (2).

Case $\texttt{C}^n \equiv$ `Cst c`: There is one new balanced subtrace, $(\texttt{Cst c},\mathrm{E}^n,\mathrm{S}^n) \Rightarrow (\texttt{Ret},\texttt{c},\mathrm{S}^n)$. Since $\mathcal{P}_e[\![\,\texttt{Ret}\,]\!]\phi\rho = \{\}$, it is trivially the case that $\mathcal{P}_e[\![\,\texttt{Cst c}\,]\!]\phi\rho \supseteq \mathcal{P}_e[\![\,\texttt{Ret}\,]\!]\phi\rho$, so property (1) is preserved. The swapping of environments and stack top elements does not affect property (2).

Case $\texttt{C}^n \equiv$ `B1-A` or `B2-A`: As for `Cst c`.

Case $\texttt{C}^n \equiv$ `If`: There are two possible transitions, one for each of the two branches of the conditional. Consider for instance $(\texttt{If},\mathrm{E},\texttt{Ret}\bullet\mathrm{T} : \texttt{C}_2\bullet\mathrm{E}_2 : \texttt{C}_3\bullet\mathrm{E}_3 : \mathrm{S}) \Rightarrow (\texttt{C}_2,\mathrm{E}_2,\mathrm{S})$. This introduces a new simple balanced trace whose sequence of instructions is $\texttt{(App (App (App C}_1\ \texttt{If) C}_2\texttt{) C}_3\texttt{)}$, $\texttt{(App (App C}_1\ \texttt{If) C}_2\texttt{)}$, $\texttt{(App C}_1\ \texttt{If)}$, …, `Ret`, `If`, where $\texttt{C}_i = \mathcal{C}[\![\,\texttt{e}_i\,]\!]\rho_c$ for $i = 1, 2, 3$. But $\mathcal{P}_e[\![\,\texttt{(App (App (App C}_1\ \texttt{If) C}_2\texttt{) C}_3\texttt{)}\,]\!]\phi\rho = \mathcal{P}_e[\![\,\texttt{if e}_1\ \texttt{e}_2\ \texttt{e}_3\,]\!]\phi\rho = \mathcal{P}_e[\![\,\texttt{e}_2\,]\!]\phi\rho \cup \mathcal{P}_e[\![\,\texttt{e}_3\,]\!]\phi\rho \supseteq \mathcal{P}_e[\![\,\texttt{e}_2\,]\!]\phi\rho = \mathcal{P}_e[\![\,\texttt{C}_2\,]\!]\phi\rho$, by definition of $\mathcal{P}_e$ on `if`, which shows that property (1) holds for the $(n+1)$-prefix. Property (2) holds because E_2 and S are the same as in an earlier state (found already in the n-prefix).

4.2.6 Correctness for letrec

In the only new kind of simple balanced trace the sequence of instructions simply consists of the `Lrc` instruction.

simple ::= ... | `Lrc`

The labelled translation of `letrec` is

$$\mathcal{C}[\![\texttt{letrec f}_1\texttt{=}^{\ell_1}\texttt{e}_1\ \ldots\ \texttt{f}_m\texttt{=}^{\ell_m}\texttt{e}_m\ \texttt{in e}_0]\!]\rho_c = \begin{array}{l}\text{let } \rho_c{}' = \texttt{f}_m : \ldots : \texttt{f}_1 : \rho_c \text{ in} \\ \text{let } \texttt{C}_i\ = \mathcal{C}[\![\texttt{e}_i]\!]\rho_c{}' \text{ for } i = 0, \ldots, m \text{ in} \\ \quad \texttt{Lrc (C}_1{}^{\ell_1}\ \ldots\ \texttt{C}_m{}^{\ell_m}\texttt{) C}_0\end{array}$$

We extend the case analysis from the proof as follows:

Case $\texttt{C}^n \equiv \texttt{Lrc}$**:** The n-prefix is extended with the transition $(\texttt{(Lrc (C}_1{}^{\ell_1}\ \ldots\ \texttt{C}_m{}^{\ell_m}\texttt{) C}_0\texttt{)},\mathrm{E},\mathrm{S}) \Rightarrow (\texttt{C}_0,\mathrm{E}',\mathrm{S})$. This introduces one new simple balanced trace of length 1. Since $\mathcal{P}_e[\![\texttt{(Lrc (C}_1{}^{\ell_1}\ \ldots\ \texttt{C}_m{}^{\ell_m}\texttt{) C}_0\texttt{)}]\!]\phi\rho = \mathcal{P}_e[\![\texttt{C}_0]\!]\phi\rho$ by definition of $\mathcal{P}_e$ on `letrec`, property (1) holds also in the $(n+1)$-prefix. The new environment E′ is ρ-safe because for every $i = 1, \ldots, m$, it holds that $\rho\ \ell_i = \mathcal{P}_e[\![\texttt{C}_i]\!]\phi\rho$ by definition of ρ in the closure analysis, and every E_i in E′ is either equal to E′ or is in E and hence is ρ-safe.

4.3 Implementation

We have an experimental implementation of the closure analysis which goes together with the implementation of L and the K machine. The implementation and its use are briefly described in Appendix B.2 which also gives an example.

4.4 Extending analyses to higher order languages

The closure analysis is used for extending essentially first order analysis methods to higher order languages. The main difference between first order and higher order program analysis lies in the handling of function applications. In a first order language, applications have form `(f e)` where `f` is the name of a defined function. A first order strictness analysis (say) can keep a table θ of the abstract value of each defined function and simply look it up as θ`f`, then combine it with the abstract value of `e` to obtain the abstract value of the application. In a higher order language, applications have the more general form $(\texttt{e}_1\ \texttt{e}_2)$ where it is not clear which function is really applied to $\texttt{e}_2$. There are now two obvious ways to proceed.

First, one could use some auxiliary analysis to find an approximation to the set of functions that $\texttt{e}_1$ could evaluate to, and use the map θ as above to get an (approximate) abstract value valid for all those functions. The abstract value of the application is the result of combining this abstract value with that of the argument. This is the closure analysis approach.

Secondly, one could let the abstract value of e_1 be *a function*, which when applied to the abstract value of e_2 gives the abstract value of the application. This is the approach taken by Burn, Hankin, and Abramsky for strictness analysis of monomorphically typed lambda calculus [11]. For an *untyped* language, it seems necessary to let the abstract value of an expression be a *pair* of those properties of the expression which are independent of its type, and the properties of the expression when applied as a function. This gives the "strictness pairs" method of Hudak and Young [35].

Below we compare the two approaches for untyped languages: the closure analysis method and the strictness pairs method.

4.4.1 Complexity of closure analysis

An upper bound on the complexity of the closure analysis can be found by considering the height of the domain CDescription^2 of (ϕ,ρ), where CDescription = Label $\to \wp$(Label). If the number of elements of Label is n, then the height of $\wp$(Label) is n, and the height of CDescription is n^2, the height of CDescription^2 is $2n^2$. Thus function $\mathcal{P}_v$ needs to make at most $2n^2$ passes over the program. The number of elementary operations in a pass is proportional to the size of the program, and every elementary operation takes time at most proportional to n. (The only non-constant time operation is in the $\mathcal{P}_v$ rule for application). We can assume also that the number n of labels (and hence lambdas and letrec-bindings) in the expanded program is proportional to the size of the program. Hence the complexity of the closure analysis is $O(p^4)$, where p is the size of the program. In conclusion, closure analysis can be done in polynomial time.

This is not fast, but tractable, and reflects the complexity of a completely unsophisticated implementation. A more serious problem is that the analysis is not easily extensible to separate compilation: it is necessary to analyse the entire program at the same time.

4.4.2 The strictness pairs method

The strictness pairs method was developed by Hudak and Young at Yale University, and is so called because it was originally used in a higher order strictness analysis for the (almost) untyped lambda calculus [35]. The strictness pairs method has subsequently been used in several papers by Hudak's colleagues for a variety of higher order program analyses. Bloss developed higher order path analysis and update analysis [6], and Goldberg developed a higher order sharing analysis [27] as well as escape analyses [28] [50].

The strictness pairs method of Hudak and Young can be described roughly as follows. Assume we have a first order (fo) analysis working on abstract values v. In the extension to a higher order (ho) analysis, an ho abstract value a is a pair (v, f) where v is an (fo) abstract value, and f is a function from abstract values to abstract values. So the ho abstract value for a function valued expression e is a pair $a = (v, f)$, called a *strictness pair*, where v describes the (strictness) properties of e itself, and f describes the (strictness) properties of the function that e evaluates to. Hudak and Young call v the "direct strictness" and f the "delayed strictness" of e.

The abstract value a of a base value will be a pair (v,serr) whose second component "serr" means "is not applicable" — it has no delayed strictness. If a abstracts a functional value, then f will map the abstract value a_i of its argument to the abstract value a_o of the result of the application. The domain of (ho) abstract values (such as strictness pairs) is $\mathrm{SP} = \mathrm{Sv} \times (\mathrm{SP} \to \mathrm{SP})$ where Sv is the domain of abstract values in the underlying first order analysis. Note that even if Sv is a lattice of finite height, SP will have infinite chains.

4.4.3 Complexity of the strictness pairs method

For this reason there is nothing that guarantees that a strictness pairs based analysis will terminate when applied to an arbitrary untyped lambda term [35, page 103–104].

However, when applied to lambda terms with some form of weak typing, the analysis *will* terminate. It is certainly *sufficient* to impose a monomorphic type discipline as in the Burn-Hankin-Abramsky framework [11] or a Milner style polymorphic one as in Abramsky's extension to that framework [1]. It is also sufficient to require that all values have *reducing type* [26], that is, no value can be applied arbitrarily many times without producing a base value. On the other hand, these restrictions are not *necessary* for the strictness pairs method to terminate, and it is not clear precisely what is necessary.

For these reasons we shall consider the worst-case complexity only in the restricted case of a monomorphically typed lambda calculus (in which case Hudak and Young's approach is much the same as Burn, Hankin, and Abramsky's).

Let A_b be the set of abstract values in the underlying first order analysis, and let $A(t)$ denote the domain of strictness pairs over type t. Then we have

$$\begin{aligned} A(\mathtt{nat}) &= A_b \times \{\mathrm{serr}\} \\ A(t_1 \to t_2) &= A_b \times (A(t_1) \to A(t_2)) \end{aligned}$$

The number of elements of $A(t_1 \to t_2)$ is $|A(t_1 \to t_2)| = |A_b| \times |A(t_2)|^{|A(t_1)|}$, and the height of $A(t_1 \to t_2)$ is $\mathit{height}(A(t_1 \to t_2)) = \mathit{height}(A_b) + |A(t_1)| \times \mathit{height}(A(t_2))$.

It is not hard to see that for $t = t_1 \to \ldots\ t_k \to t'$, the height of domain $A(t)$ is approximately proportional to the product of the sizes of $A(t_j)$ for each t_j. In particular, for a k-ary function all of whose arguments have the same type (*e.g.*, base type), the height is exponential in k.

Moreover, if we define the *rank* $r(t)$ of a type by $r(\mathtt{nat}) = 0$, and $r(t_1 \to t_2) = 1 + r(t_1)$, we see that the height of $A(t)$ is at least n to the n'th to the n'th $\ldots$, where the stack of n's is $r(t)$ high, and n is the number of elements in the first order abstract domain A_b.

The number of iterations needed (in an analysis) to find the abstract value a for an expression is bounded by the height of the abstract domain. Thus the worst case complexity is outrageous when the types get complex. However, this does not mean that the complexity of the analysis grows very much with the size of a program, so long as the types and arities involved remain bounded.

A more practical problem in the strictness pairs approach is the necessity to construct, compare, and apply representations of (higher order) functions during the analysis. This is tolerable for small underlying abstract domains, such as the two-point domain of strictness analysis, but for *e.g.* path analysis it was so expensive that Bloss did not consider full higher order path analysis implementable after all [6, page 70].

4.4.4 Applicability to analysis of data structures

The strictness pairs method as presented by Hudak and Young does not explicitly handle data structures, but it could conceivably do so by encoding them as functions [35, Section 4.7]. However, it is not clear whether this actually works for recursively defined infinite data structures, the hallmark of lazy languages.

To ensure termination of the strictness pairs method, the second component of a strictness pair must always eventually produce "serr" when applied sufficiently many times [35, pp. 103–104]. This amounts to requiring all values to have reducing type, but the encoding of an infinite list such as `ones = (Cons 1 ones)` or `from n = (Cons n (from (n+1)))` does not have reducing type. This is seen by taking the tail of the infinite lists.

However, while the "reducing type" requirement is *sufficient* for termination, the discussion in [35] does not make it entirely clear whether it is really *necessary*.

Moreover, even if the analysis *is* applicable to data structures encoded as functions, then the complexity may be high. The reason is that the functions resulting from the encoding have types of high rank. The type of a function `f xs = ...` which takes a list `xs` of base values as input, has rank 2. Hence the height to the domain of abstract values for `f` is at least $O(n^n)$ where n is the size of the domain of abstract values over base types. If `xs` is a list of functions, or a list of functions on lists, and so on, then the rank is even higher, and the height of the abstract value domain grows enormously. Note that this does not affect the complexity in terms of the size of the program.

So it is uncertain whether the Hudak and Young strictness pair method can be used to analyse data structures encoded as functions.

4.4.5 Comparison

In conclusion, the "strictness pairs" method is precise, expensive, requires programs to be slightly typed, and may not be applicable to analysis of data structures encoded as higher order functions.

The closure analysis method, on the other hand, is reasonably fast, imprecise, requires no typing, and is applicable to analysis of data structures encoded as higher order functions, albeit possibly with considerable loss of precision.

We claim that for many purposes the closure analysis approach is both feasible and usably precise, cf. Section 4.1.3. In fact, it is sufficiently good to be used in Bondorf and Danvy's rather substantial self-applicable partial evaluator Similix for Scheme [10].

To obtain better analyses for data structures one would probably need to make use of the type structure in the language.

Chapter 5

Usage Interval Analysis

This chapter describes a so-called usage interval analysis. The analysis computes lower and upper bounds on the number of times an expression $\mathtt{e}_i$ may be evaluated when the containing expression $\mathtt{e} \equiv (\ldots \mathtt{e}_i \ldots)$ is evaluated once. This provides approximate information on the number of times a function application may evaluate its argument expression, assuming that parameter passing is call by name.

The results of this analysis are used for two purposes. The lower bound provides strictness information, and the upper bound provides sharing information. If the lower bound is greater than or equal to one, then the argument *must* be evaluated, and one may as well evaluate the argument before the function body. If the upper bound is less than or equal to one, then the argument cannot be used again after its first use, and no sharing mechanism is needed to obtain call by need or lazy evaluation.

The lower bound or strictness optimization can be said to replace call by need with call by value [47]. Similarly, the upper bound or sharing optimization can be said to replace call by need with call by name [38].

The usage interval analysis works on the lazy language L introduced in Chapter 3. Originally it was developed for Guy Argo's abstract machine G-TIM [3] [4], a version of Fairbairn and Wray's Three Instruction Machine. The present version of the analysis has been simplified and extended to cover data structures, lambdas, and local recursive (`letrec`) definitions.

5.1 Usage interval analysis

The purpose of the usage interval analysis is to find lower and upper bounds on the number of times an argument to a lambda may be evaluated if the language implementation were to use call by name instead of call by need (*i.e.*, were not lazy, just non-strict).

The usage interval analysis subsumes simple strictness analysis and sharing analysis: If the lower bound on the usage interval of the argument is greater than *Zero*, then the combinator is strict in that argument. If the upper bound is *Many*, then the argument may be shared, otherwise it cannot be.

5.1.1 Usage Counts and Usage Intervals

The *usage count* for a variable in an evaluation is a "number" in the set $U = \{Zero, One, Many\}$, where *Many* represents any number greater than 1. The set of usage counts is ordered $Zero < One < Many$. Addition and multiplication of usage counts can be defined in the obvious way. For example, $One + One = Many$, $Many + u = Many$ for all $u \in U$, $Zero \times u = Zero$ for all $u \in U$, *etc.* The minimum and maximum of a set of usage counts are defined in the obvious way.

A *usage interval* is a set $\{u \in U | u_{min} \leq u \leq u_{max}\}$ of usage counts, and is denoted by $[u_{min}, u_{max}]$ where $u_{min}, u_{max} \in U$. The set UI of usage intervals can be ordered by set inclusion (which makes it a lattice). The least upper bound $[u_{11}, u_{12}] \sqcup [u_{21}, u_{22}]$ of two usage intervals is the least usage interval containing both.

Addition and multiplication of usage counts extend to usage intervals straightforwardly: $[u_{11}, u_{12}] + [u_{21}, u_{22}] = [u_{11} + u_{21}, u_{12} + u_{22}]$, and $[u_{11}, u_{12}] \times [u_{21}, u_{22}] = [u_{11}u_{21}, u_{12}u_{22}]$.

These operations are used for mimicking the effect of combining expressions syntactically: parallel uses $(+)$, application of one to the other $(\times)$, and alternative uses $(\sqcup)$. Let us consider an example of usage intervals.

Example 1 In the L program

```
letrec  d w       = (+ w w)
        g x y z v = (if (= z 0)
                        (+ (d x) y)
                        (+ y (+ z v)))
in      (g e1 e2 e3 e4)
```

the variables have the following usage intervals: `w` has [*Many*,*Many*], `x` has [*Zero*,*Many*], `y` has [*One*,*One*], `z` has [*One*,*Many*], and `v` has [*Zero*,*One*].

To illustrate the effect of higher order functions, consider the program

```
letrec f     x y = (+ x y)
       twice g z = (g (g z))
in     (twice (f 7) 9)
```

Obviously `y` has usage interval [*One*,*One*]. To find the usage interval of `g` and `z` we need to know which functions `g` may be bound to. The closure analysis will tell us that `g` can be bound to `f` applied to one argument v, so `(g (g z))` is really `(f` v `(f` v `z))` for some value v. With our knowledge about the usage interval of `y` we find that `z` has usage interval [*One*,*One*] and `g` has usage interval [*Many*,*Many*].

This implies that the partial application `(f 7)` may be shared (used more than once). But then the first argument `x` of `f` may be shared too, so its usage interval is [*One*,*Many*] and not [*One*,*One*] as one might expect: the sharability of a lambda expression propagates to its free variables (here manifested by the sharability of a partial application propagating to its arguments). □

5.1.2 Usage interval analysis for core L

For a given core L program $\mathtt{e}_p$, the usage interval analysis constructs a usage description ψ and a sharing description χ. The usage count analysis uses the closure descriptions ϕ and ρ which must be computed by a previous closure analysis of $\mathtt{e}$ (Section 4.1.1).

$$\psi, \chi \; : \; \text{UDescription} \; = \; \text{Label} \rightarrow UI \qquad\qquad \text{Usage description}$$

The *usage description* ψ and the *sharing description* χ found by the analysis are intended to satisfy

$$\begin{aligned} \psi\ \ell &= \text{the usage interval of (lambda- or letrec-) variable } \mathtt{x}^\ell \text{ in its scope} \\ \chi\ \ell &= \text{the usage interval of the lambda expression } \lambda^\ell\mathtt{x}.\mathtt{e} \end{aligned}$$

Thus $\chi\ell$ is a safe approximation to the number of times the lambda will be applied.

The usage interval analysis consists of two analysis functions. The purpose of function $\mathcal{U}_e$, called the *usage analysis function*, is to compute the usage interval of a variable in an expression: $\mathcal{U}_e[\![\, \mathtt{e}_1 \,]\!]\psi\chi\ell'$ is the usage interval of variable $\mathtt{x}^{\ell'}$ in expression $\mathtt{e}_1$, when $\mathtt{e}_1$ is evaluated once.

The purpose of function $\mathcal{U}_v$, called the *usage propagation function*, is to compute the usage interval of every lambda $\lambda^{\ell'}\mathtt{x}.\mathtt{e}_1$ in the program $\mathtt{e}_p$: $\mathcal{U}_v[\![\, \mathtt{e}_p \,]\!]\psi\chi\ell'$ is the union of the usage intervals of all those (lambda- or letrec-) variables of which $\lambda^{\ell'}\mathtt{x}.\mathtt{e}_1$ is a possible value, and of the usage interval $[One,One]$ if the lambda expression appears in the function position of an application. In that position it will be applied exactly once when the containing expression is evaluated once.

The usage analysis function $\mathcal{U}_e$ is defined as follows:

$$\begin{aligned}
&\mathcal{U}_e : \text{Expression} \rightarrow \text{UDescription} \rightarrow \text{UDescription} \rightarrow \text{UDescription} \\
&\mathcal{U}_e[\![\, \lambda^\ell\mathtt{x}.\mathtt{e} \,]\!]\psi\chi\ell' && = [Zero,Zero] \quad \text{if } \ell{=}\ell' \\
& && = \chi\ell \times \mathcal{U}_e[\![\, \mathtt{e} \,]\!]\psi\chi\ell' \quad \text{otherwise} \\
&\mathcal{U}_e[\![\, \mathtt{x}^\ell \,]\!]\psi\chi\ell' && = [One,One] \quad \text{if } \ell{=}\ell' \\
& && = [Zero,Zero] \quad \text{otherwise} \\
&\mathcal{U}_e[\![\, (\mathtt{e}_1\ \mathtt{e}_2) \,]\!]\psi\chi\ell' && = \mathcal{U}_e[\![\, \mathtt{e}_1 \,]\!]\psi\chi\ell' \\
& && \quad + \mathcal{U}_e[\![\, \mathtt{e}_2 \,]\!]\psi\chi\ell' \times \bigsqcup \{\psi\ell \mid \ell{\in}\mathcal{P}_e[\![\, \mathtt{e}_1 \,]\!]\phi\rho\} \\
&\mathcal{U}_e[\![\, (\mathtt{A}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n) \,]\!]\psi\chi\ell' && = \textstyle\sum_{j=1}^{n} \mathcal{U}_e[\![\, \mathtt{e}_j \,]\!]\psi\chi\ell' \\
&\mathcal{U}_e[\![\, \mathtt{if}\ \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{e}_3 \,]\!]\psi\chi\ell' && = \mathcal{U}_e[\![\, \mathtt{e}_1 \,]\!]\psi\chi\ell' \\
& && \quad + \mathcal{U}_e[\![\, \mathtt{e}_2 \,]\!]\psi\chi\ell' \sqcup \mathcal{U}_e[\![\, \mathtt{e}_3 \,]\!]\psi\chi\ell' \\
&\mathcal{U}_e\left[\!\!\left[\begin{array}{l}\mathtt{letrec}\ \mathtt{f}_1\,\mathtt{=e}_1\ \ldots \\ \qquad\quad \mathtt{f}_m\mathtt{=e}_m\ \ \mathtt{in}\ \mathtt{e}_0\end{array}\right]\!\!\right]\psi\chi\ell' && = [Zero,Zero] \quad \text{if } \ell' \in \{\ell_1,\ldots,\ell_m\} \\
& && = \mathcal{U}_e[\![\, \mathtt{e}_0 \,]\!]\psi\chi\ell' + \textstyle\sum_{j=1}^{m} \psi\ell_j{\times}\mathcal{U}_e[\![\, \mathtt{e}_j \,]\!]\psi\chi\ell' \\
& && \qquad \text{otherwise}
\end{aligned}$$

These analysis equations are justified as follows.

In a lambda expression $\lambda^{\ell}\texttt{x.e}$, variable $\texttt{y}^{\ell'}$ cannot be used at all if $\ell{=}\ell'$ (*i.e.*, x and y are the same variable), so in this case its usage interval is [*Zero*,*Zero*]. Otherwise, if $\ell{\neq}\ell'$, then the usage interval of y in $\lambda^{\ell}\texttt{x.e}$ is $u_\ell \times u$ where u_ℓ is the usage interval of the lambda, and u is the usage interval of $\texttt{y}^{\ell'}$ in the lambda body e.

In expression $\texttt{x}^{\ell}$, variable $\texttt{y}^{\ell'}$ is used exactly once if $\ell{=}\ell'$ (*i.e.*, the variables are the same), otherwise exactly zero times.

In an application $(\texttt{e}_1\ \texttt{e}_2)$, variable $\texttt{y}^{\ell'}$ may be used first to evaluate $\texttt{e}_1$ to some function $\lambda^{\ell}\ldots$, then it may be used to evaluate $\texttt{e}_2$ when that function is applied to $\texttt{e}_2$. Its use in this application is the product of its use in the argument and the number $\psi\ell$ of times the function evaluates its argument. For this it is safe to take the least upper bound of the usage intervals of all functions that $\texttt{e}_1$ can evaluate to.

In a (strict) base function application, the number of uses of $\texttt{y}^{\ell'}$ is the sum of its use in the arguments.

In a conditional expression $\texttt{if}\ \texttt{e}_1\ \texttt{e}_2\ \texttt{e}_3$, variable $\texttt{y}^{\ell'}$ is used first to evaluate the root expression $\texttt{e}_1$ to obtain a truth value, then either of the two branches will be taken. It is safe to take the least upper bound of the usage intervals of y in the two branches.

In a local recursive definition $\texttt{letrec}\ \texttt{f}_1\texttt{=}^{\ell_1}\texttt{e}_1\ \ldots\ \texttt{f}_m\texttt{=}^{\ell_m}\texttt{e}_m\ \texttt{in}\ \texttt{e}_0$, variable $\texttt{y}^{\ell'}$ cannot be used at all if there is a j such that $\ell'{=}\ell_j$ (*i.e.*, variables $\texttt{y}^{\ell'}$ and $\texttt{f}_j{}^{\ell_j}$ are the same), so in this case the usage interval is [*Zero*,*Zero*]. Otherwise the usage interval is the sum of the uses in each of the definitions and in the body of the letrec definition. The use of y in a definition $\texttt{f}_j\texttt{=}^{\ell_j}\texttt{e}_j$ is the product of its use in the right hand side $\texttt{e}_j$ and the usage interval $\psi\ell_j$ of the bound variable $\texttt{f}_j$.

This should suffice to explain the usage interval analysis equations.

The usage propagation function $\mathcal{U}_v$ is defined by the following equations:

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{U}_v : \text{Expression} \rightarrow \text{UDescription} \rightarrow \text{UDescription}} \\
\mathcal{U}_v[\![\, \lambda^{\ell}\texttt{x.e} \,]\!]\psi\ell' & = & \bigsqcup \{\ \psi\ell \mid \ell' \in \rho\ell\ \} \sqcup \mathcal{U}_v[\![\, \texttt{e} \,]\!]\psi\ell' \\
\mathcal{U}_v[\![\, \texttt{x}^{\ell} \,]\!]\psi\ell' & = & [\mathit{Many},\mathit{Zero}] \\
\mathcal{U}_v[\![\, (\texttt{e}_1\ \texttt{e}_2) \,]\!]\psi\ell' & = & \mathcal{U}_v[\![\, \texttt{e}_1 \,]\!]\psi\ell' \sqcup \mathcal{U}_v[\![\, \texttt{e}_2 \,]\!]\psi\ell' \\
 & & \sqcup \bigsqcup \{\ [\mathit{One},\mathit{One}] \mid \ell' \in \mathcal{P}_e[\![\, \texttt{e}_1 \,]\!]\phi\rho\ \} \\
\mathcal{U}_v[\![\, (\texttt{A}\ \texttt{e}_1\ \ldots\ \texttt{e}_n) \,]\!]\psi\ell' & = & \bigsqcup_{j=1}^{n} \mathcal{U}_v[\![\, \texttt{e}_j \,]\!]\psi\ell' \\
\mathcal{U}_v[\![\, \texttt{if}\ \texttt{e}_1\ \texttt{e}_2\ \texttt{e}_3 \,]\!]\psi\ell' & = & \bigsqcup_{j=1,2,3} \mathcal{U}_v[\![\, \texttt{e}_j \,]\!]\psi\ell' \\
\mathcal{U}_v\left[\!\!\left[\begin{array}{l}\texttt{letrec}\ \texttt{f}_1\texttt{=e}_1\ \ldots \\ \qquad\quad \texttt{f}_m\texttt{=e}_m\ \texttt{in}\ \texttt{e}_0\end{array}\right]\!\!\right]\psi\ell' & = & \bigsqcup_{j=0}^{m} \mathcal{U}_v[\![\, \texttt{e}_j \,]\!]\psi\ell' \sqcup \bigsqcup_{j=1}^{m} \left(\bigsqcup \{\ \psi\ell_j \mid \ell' \in \rho\ell_j\}\right)
\end{array}
$$

The $\mathcal{U}_v$ analysis equations are justified as follows.

If lambda $\lambda^{\ell'}\ldots$ can be bound to variable $\texttt{x}^{\ell}$ (that is, if $\ell' \in \rho\ell$), where $\texttt{x}^{\ell}$ has usage interval $\psi\ell$, then $\psi\ell$ is a subset of the usage interval of lambda $\lambda^{\ell'}\ldots$.

In a variable $\texttt{x}^{\ell}$, there is no binding of lambda ℓ' and hence no contribution to its usage interval (the empty usage interval [*Many*,*Zero*] is the identity element for least upper bound of usage intervals).

If lambda $\lambda^{\ell'}\ldots$ is a possible value of the subexpression $\mathtt{e}_1$ in the function position of an application $(\mathtt{e}_1\ \mathtt{e}_2)$, then it will be used exactly once in an evaluation of the application. Moreover, it may be used in the evaluation of each of the subexpressions.

In a full base function application and in a conditional, a lambda may be used only in the subexpressions.

In a local recursive definition, if lambda $\lambda^{\ell'}\ldots$ is a possible value of `letrec` variable $\mathtt{f}_j$ which has usage interval $\psi\ell_j$, then $\psi\ell_j$ is a subset of the usage interval of lambda $\lambda^{\ell'}\ldots$. In addition, the lambda may be used in the right hand sides $\mathtt{e}_i$ and in the body $\mathtt{e}_0$ of the `letrec`.

5.1.3 Usage interval analysis of L programs

The whole point of the functions $\mathcal{U}_e$ and $\mathcal{U}_v$ is to compute the pair of descriptions $(\psi,\chi) \in$ UDescription^2 which gives the usage interval of every variable $\mathtt{x}^\ell$ and of every lambda $\lambda^\ell\mathtt{x}.\mathtt{e}_1$ in a given core L program $\mathtt{e}_p$.

Let such a program $\mathtt{e}_p$ be given. The desired pair of descriptions is found as the least solution to the set of simultaneous equations

$$
\begin{aligned}
\psi\ell &= \mathcal{U}_e[\![\,\mathtt{e}_1\,]\!]\psi\chi\ell && \text{for all } \lambda^\ell\mathtt{x}.\mathtt{e}_1 \text{ in } \mathtt{e}_p\\
\chi\ell &= \mathcal{U}_v[\![\,\mathtt{e}_p\,]\!]\psi\ell && \text{for all } \lambda^\ell\mathtt{x}.\mathtt{e}_1 \text{ in } \mathtt{e}_p\\
\psi\ell_i &= \mathcal{U}_e[\![\,\mathtt{e}_0\,]\!]\psi\chi\ell_i + \textstyle\sum_{j=1}^{m}\psi\ell_j\times\mathcal{U}_e[\![\,\mathtt{e}_j\,]\!]\psi\chi\ell_i\\
&\quad \text{for all } \mathtt{letrec}\ \mathtt{f}_1\mathtt{=}^{\ell_1}\mathtt{e}_1\ \ldots\ \mathtt{f}_m\mathtt{=}^{\ell_m}\mathtt{e}_m\ \mathtt{in}\ \mathtt{e}_0 \text{ and for all } i=1,\ldots,m
\end{aligned}
$$

5.1.4 Complexity of usage interval analysis

An upper bound on the complexity of the usage interval analysis can be found by considering the height of the domain UDescription^2, where UDescription = Label $\to UI$. With the inclusion ordering, UI is a lattice of height 3. Assuming that the number of elements in Label is n, UDescription is a lattice of height $3n$ (with the pointwise inclusion ordering), and UDescription^2 has height $6n$. Both $\mathcal{U}_e$ and $\mathcal{U}_v$ are monotonic functions, so the least fixed point can be found in at most $6n$ passes over the L program. Every pass can be done in time proportional to the square of the size of the program. (There is a non constant time set operation in the $\mathcal{U}_e$ equation for application). Assuming that the number of labels (*i.e.*, of lambdas and letrec-bindings) is proportional to the size of the program, we find that the complexity of the usage interval analysis is roughly $O(p^3)$ where p is the size of the L program.

5.1.5 Analysing data structures

To analyse an L program $\mathtt{e}_p$ whose result is a lazy data structure, we need to provide an initial context for the program. This initial context must drive the lazy evaluation in a way which simulates the way printing does it.

The functions `PN` and `PN'` for printing `Nil`, and the functions `PC` and `PC'` for printing `Cons` were shown in Section 3.4.6, and we observed that when disregarding the actual printing, the primed and non-primed functions behave the same.

Thus we introduce two functions `printnil` and `printcons` which simulate `PN/PN'` and `PC/PC'`, and analyse the application (`e`$_p$ `printnil printcons`) of the program `e`$_p$ to these functions. This artificial context is removed again before we use the analysis results in the optimizing compilation. The print simulators are defined as follows:

```
printnil  = λ x.x
printcons = λ x.λ y. (A (x printnil printcons)
                        (y printnil printcons))
```

The function `A` is any strict base function. It is easy to see that `printnil` behaves like `PN` or `PN'` when applied to a non-base value. Similarly, one can see that `printcons` behaves like `PC` or `PC'`, where the `R1` and `B2-A` instructions do the work of the `PCR` instruction.

However, having to add this context is not really neat, and makes sense only for programs whose result is a data structure. In a typed language this could be given a more convincing treatment.

5.1.6 Related work

Hughes and Wray describe a method to estimate the number of times an expression will be evaluated under call by name, within Hughes's general backwards analysis framework [38, Section 4.2]. This has the advantage of admitting analysis of data structures as well as higher order functions.

Goldberg presented a sharing analysis, similar to the usage interval analysis, for a higher order untyped language without data structures [27]. Jensen and Mogensen presented a similar analysis for compile time garbage collection, first for a first order language with data structures, then an extension to higher order programs by means of a closure analysis [40].

Goldberg's analysis is of the "strictness pairs" variety as discussed in Section 4.4, and thus is very expensive for higher order functions. Higher order analyses in the framework of Hughes seem to have roughly the same complexity as Goldberg's, but this has not been investigated in detail.

5.2 Optimizing call by need to call by name

Lazy evaluation is expensive because suspensions must be created for values which may or may not be needed, markers must be pushed before entering argument expressions, tests for markers must be made, and finally, suspensions must be overwritten. This section and the next one describe two ways to improve on lazy (call by need) evaluation. The first one is to replace call by need with call by name, using an *upper bound* on the number of times a function argument will be evaluated. The second one is to replace call by need with call by value, using a *lower bound* on the number of times a function argument will be evaluated. Both kinds of analysis information are provided by the usage interval analysis presented above.

The lazy parameter passing mechanism "call by need" can be replaced with "call by name" without loss of efficiency when the argument is known to be evaluated at most once. The upper bounds computed by the usage interval analysis provide precisely the information we need for this optimization. Replacing call by need by call by name cannot affect the *result* (extensional behaviour) of a program, but when inappropriately applied it may affect *efficiency* very considerably by destroying laziness.

5.2.1 Call by name in the K+ machine

In this section we shall describe an improved compilation of core L programs into instructions for an improved K machine, called the K+ machine. The compilation algorithm uses the upper bound information computed by the usage interval analysis.

We introduce two new instructions: Unsharable variable `(Var# i)` and unsharable lambda `(Lam# C)`.

A variable is *unsharable* if its usage interval $\psi\ell$ does not contain *Many* (*i.e.*, the upper bound is not *Many*). A closure bound to an unsharable variable cannot be accessed more than once, and it is not necessary to push a marker <p> before entering the closure.

A lambda is unsharable if its usage interval $\chi\ell$ does not contain *Many* (*i.e.*, the upper bound is not *Many*). This means the lambda can only be bound to unsharable variables, so there will never be a marker on the stack top when the lambda takes its argument off the stack. Hence an unsharable lambda need not check for markers.

The compilation rules for lambdas and variables in core L (summarized in Section 3.7) are extended as follows:

$$
\begin{array}{llll}
\mathcal{C}[\![\, \mathtt{x}^{\ell}{}_i \,]\!][\mathtt{x}_0,\ldots,\mathtt{x}_i,\ldots,\mathtt{x}_n] & = \mathtt{Var\#}\ i & & \text{if } \mathit{Many} \notin \psi\ell \\
 & = \mathtt{Var}\ i & & \text{otherwise} \\
\mathcal{C}[\![\, \lambda^{\ell}\mathtt{x}.\mathtt{e} \,]\!]\rho_c & = \mathtt{Lam\#}\ \mathcal{C}[\![\, \mathtt{e} \,]\!](\mathtt{x}{:}\rho_c) & & \text{if } \mathit{Many} \notin \chi\ell \\
 & = \mathtt{Lam}\ \mathcal{C}[\![\, \mathtt{e} \,]\!](\mathtt{x}{:}\rho_c) & & \text{otherwise}
\end{array}
$$

The K+ machine rules for the new instructions are as follows. All other instructions are as in the K machine (summarized in Section 3.7).

Code C	Env. E	Stack S		Code C	Env. E	Stack S	
Var# i	E	S	⇒	$\mathtt{C}_i$	E_i	S	where $\mathtt{C}_i \bullet \mathrm{E}_i = \mathrm{E}[i]$
Lam# C	E	u:S	⇒	C	u:E	S	

The introduction of unsharable variables `Var#` makes further optimizations possible. First we observe that an application `(App` $\mathtt{C}_1$ `(Var# k))` to an unsharable variable would pair the `Var#` instruction with the environment E and push the closure `Var# k`•E onto the stack. Only later (if ever) would the closure be executed and extract the value $\mathtt{C}_k \bullet \mathrm{E}_k = \mathrm{E}[k]$ from E and enter it. It is probably better to extract the closure $\mathtt{C}_k \bullet \mathrm{E}_k$ immediately and put it onto the stack to be entered later on (if ever). There is a trade-off between the time it takes to (maybe needlessly) extract $\mathtt{C}_k \bullet \mathrm{E}_k$ from E, the time it would take to construct `Var# k`•E, and the likelihood that the variable is used at all. However, extracting the closure immediately has the added advantage that it does not keep all the other components of E "live" while `Var# k`•E is on the stack, so the optimization may reduce the heap residency (space consumption).

However, the optimization is admissible only because the variable is unsharable; were it sharable, then the closure might be entered and evaluated several times. Concretely, this would be manifested by the need to push a marker <p> before entering $\mathtt{C}_k \bullet \mathrm{E}_k$.

We introduce a new special instruction `App# C k` which loads the closure $\mathtt{C}_k \bullet \mathrm{E}_k$ bound to variable $\mathtt{x}_k$ directly from the environment E, and optimize the K+ machine code by replacing occurrences of `(App C (Var# k))` by `(App# C k)`.

Code C	Env. E	Stack S		Code C	Env. E	Stack S	
App# $\mathtt{C}_1$ k	E	S	$\Rightarrow$	$\mathtt{C}_1$	E	$\mathtt{C}_k \bullet \mathrm{E}_k$: S	where $\mathtt{C}_k \bullet \mathrm{E}_k = \mathrm{E}[k]$

Unsharable variables can be used also for `letrec`-bound textual functions (lambdas), or more generally, for variables known to be bound only to objects already in whnf. There is no need to push a marker <p> before entering such a value, since it would immediately overwrite itself to no avail. However, this does not mean that a `letrec`-bound lambda should not check for markers, for it is quite possible that it is the whnf of a closure bound to some sharable variable which needs to be overwritten.

It is easy to change the compile time environment ρ_c to distinguish such (`letrec`-bound function-) variables and make the compilation generate `Var#` instructions for them.

5.2.2 Related work

Hughes and Wray's analysis described above was developed for optimizing call by need to call by name [38, Section 4.2].

A paper by Fairbairn on "removing redundant laziness" presents the idea of avoiding laziness overhead when an expression can be used at most once, but does not give any analysis [21].

Peyton Jones uses an intermediate language STG which distinguishes updatable and non-updatable closures. He calls analyses to introduce update annotations *update analysis*, but does not give such an analysis [52, Section 4.2]. Non-updatable closures are more closely related to our notion of unsharable variable than to our notion of unsharable lambda, but the analogy is not exact. Our analysis does not treat closures individually, it works only with the variables they can be bound to and the lambdas (whnf's) they can evaluate to. In our framework it is the responsibility of the variables and the lambdas to make sure closures are overwritten when needed. Thus the sharing analysis and optimization presented here are not readily applicable in an update analysis.

5.3 Optimizing call by need to call by value

As already mentioned, lower usage bounds give strictness information. However, the K machine is ill-suited to exploit such information. The only strictness optimization possible in the K machine is to evaluate the function argument to whnf at application time ("force it"), and treat the formal parameter as unsharable (`Var#`), so that no marker is pushed before entry to the variable's value. The variable is necessarily bound to a whnf, and there is no point in pushing a marker before entering a whnf — that would cause it to needlessly overwrite itself immediately.

Note that the value of the variable necessarily is a closure still: this is the only way to represent a value in the K machine. In more realistic machines, strictness gives the opportunity to treat base values as unboxed, that is, non-closures, or "real" machine integers, *etc.*

Unfortunately, forcing the evaluation of an expression is quite complicated for an expression whose result is not of base type. This is because there is no simple way to get control back from an expression which evaluates to a function. On the other hand, base values always activate the stack top element, so we can just let the stack top element be a function which takes and stores the evaluated base value.

5.3.1 Call by value in the K+ machine

We have experimented with strictness optimizations for base value arguments and have introduced two new instructions for the K+ machine. These are variants of lambda, `Sla` for *strict lambda*, and `Sla#` for strict unsharable lambda. Each takes its argument suspension off the stack top, puts an unsharable lambda `Lam# C` on the stack top, and enters the argument suspension. This evaluates to a base value whnf `Ret`•v which activates the lambda `Lam# C` on the stack top.

Although the language L is untyped, we know from the closure analysis that if $\rho\ell = \{\}$, then the lambda $\lambda^\ell\mathtt{x}.\mathtt{e}$ can be applied only to base type arguments. Hence the revised compilation scheme for lambdas is:

$$\begin{array}{rcll}
\mathcal{C}[\![\,\lambda^\ell\mathtt{x}.\mathtt{e}\,]\!]\rho_c & = & \mathtt{Sla\#}\,\mathcal{C}[\![\,\mathtt{e}\,]\!](\mathtt{x}{:}\rho_c) & \text{if } \mathit{Zero} \notin \psi\ell,\ \mathit{Many} \notin \chi\ell \text{ and } \rho\ell = \{\} \\
 & = & \mathtt{Sla}\,\mathcal{C}[\![\,\mathtt{e}\,]\!](\mathtt{x}{:}\rho_c) & \text{if } \mathit{Zero} \notin \psi\ell \text{ and } \rho\ell = \{\} \\
 & = & \mathtt{Lam\#}\,\mathcal{C}[\![\,\mathtt{e}\,]\!](\mathtt{x}{:}\rho_c) & \text{if } \mathit{Many} \notin \chi\ell \\
 & = & \mathtt{Lam}\,\mathcal{C}[\![\,\mathtt{e}\,]\!](\mathtt{x}{:}\rho_c) & \text{otherwise}
\end{array}$$

The new instructions are interpreted as follows in the K+ machine:

Code C	Env. E	Stack S		Code C	Env. E	Stack S	
`Sla# C`	E	C_1•E_1 : S	⇒	C_1	E_1	(`Lam# C`•E) : S	
`Sla C`	E	<p> : S	⇒	`Sla C`	E	S	and H[p] := `Sla C`•E
`Sla C`	E	C_1•E_1 : S	⇒	C_1	E_1	(`Lam# C`•E) : S	

Note that a strict lambda need not put a marker on the stack before evaluating its argument. The argument whnf, which is a base value `Ret`•v, will activate the closure `Lam# C`•E on the stack top, and will be bound in the environment E.

Because of the necessity to construct a new closure on the stack, the "optimization" is worthwhile only when the argument is likely to be used *more* than once. Thus the compilation rules above could be modified to require that the usage interval $\psi\ell$ of $\mathtt{x}^\ell$ does contain (only) *Many*.

5.3.2 Related work

There is much theoretical work on optimizing call by need to call by value, and the supporting so-called strictness analysis, starting with Mycroft's thesis [47].

Strictness analysis for higher order (monomorphically) typed languages was done by Burn, Hankin, and Abramsky [11]; and for (almost) untyped functions by Hudak and Young [35].

Wadler gave a strictness analysis for lazy lists [72]. Hughes suggested that strictness analysis of data structures should be understood as abstract interpretation of continuations [37]. Subsequently, Wadler and Hughes used projections to simplify the presentation of data structure strictness [73]. All these analyses are vastly more precise than the strictness analysis given here.

Fairbairn and Wray give a strictness detection algorithm for "second order" functions without data structures, which bears some similarity to this work [22]. Their analysis has four "usages" called Strict, Lazy, Dangerous, and Absent, which correspond respectively to usage intervals containing *One* but not *Zero*; being [*Zero*,*Many*]; being empty, and containing only *Zero*. The paper cited does not describe the extension to second order languages.

In Section 6.8 below we outline a backwards strictness analysis for data structures in a first order language. That analysis takes the types of expressions into account, is more expensive, and would give considerably better results than those of this chapter.

5.4 Experiments

This section describes a few experiments made with an implementation of the usage count analysis. The experimental implementation of the usage count analysis is listed in Appendix B.3. The optimized compilation to the K+ machine and the interpreter for the K+ machine, are described in Appendix B.4. These appendices also briefly outline how to use the implementation and give an example of its use.

5.4.1 Some measurements

The implementation of the analyses and the K+ machine have been run with three different levels of optimization as described shortly. In all cases, `letrec` bound lambdas are bound to unsharable variables as suggested at the end of Section 5.2.1.

- Without optimizations.
- With sharing optimizations (Section 5.2)
- With sharing and strictness optimizations (Sections 5.2 and 5.3)

First we give the cpu time in seconds (excluding garbage collection) for a number of L expressions. The L functions used in the expressions are defined in Appendix B.5. These timings were obtained from Chez Scheme running on a Sun Sparcstation 2 and suffer some variation from run to run.

Optimizations	None	Sharing	Sharing and strictness
`take 100 primes`	13.65	13.09	13.28
`hd (drop 100 primes)`	13.16	12.46	13.82
`nth 1000 ones`	1.46	1.33	1.43
`nth 1000 (scan * 1 (from 1))`	6.25	5.30	5.76
`foldl * 1 (take 1000 (from 1))`	6.06	5.76	6.04
`(lam x (+ x x)) (fact 100)`	0.14	0.13	0.15
`(lam x (+ x 1)) (fact 100)`	0.15	0.14	0.16
`fact 1000`	3.50	3.32	3.41
`take 100 fibs`	0.69	0.71	0.70
`take 7 (repeat (fact 100))`	0.27	0.24	0.23

It is immediate that the sharing optimization reduces the run time for most programs with 3 to 10 %. The strictness optimization seems to nullify most of the gains from the sharing optimization. This should come as no great surprise because of the clumsy way strictness has to be obtained in the K+ machine.

Below we study the effect of the optimizations on the number of marker checks (as done by the instructions `Ret`, `Lam`, and `Sla`), and the number of marker updates (that is, the number of markers actually encountered by the checks).

Optimizations	None		Sharing		Sharing and strictness	
	Check	Update	Check	Update	Check	Update
`take 100 primes`	92152	66535	73342	48471	73342	48471
`hd (drop 100 primes)`	91455	65841	55223	47977	67031	36515
`nth 1000 ones`	11013	6007	7007	3004	8008	1003
`nth 1000 (scan * 1 (from 1))`	29014	19002	13005	9000	16006	6999
`foldl * 1 (take 1000 (from 1))`	29017	21004	13006	11000	16008	8999
`(lam x (+ x x)) (fact 100)`	809	303	808	303	910	0
`(lam x (+ x 1)) (fact 100)`	809	302	808	301	910	0
`fact 1000`	8006	3001	8006	3001	9007	0
`take 100 fibs`	4267	3161	3260	2077	3361	1876
`take 7 (repeat (fact 100))`	945	392	880	340	996	24

As can be seen, the sharing optimization leads to a considerable reduction in checks and updates except for benchmarks which work exclusively with base values, such as those involving `fact`. In general, the strictness analysis further reduces the number of updates, and in the base value cases, it eliminates *all* updates. On the other hand, it increases the number of checks. This is because base values are still boxed, so everything done with a base value involves a marker check. The results are rather satisfactory, though. Had we used a machine that could exploit strictness information, good speed-ups would have been obtained.

5.4.2 Imprecision in the analysis of data structures

Unfortunately, the usage interval analysis as presented above turns out to give rather imprecise results for analysis of data structures. This is due to the simple-minded treatment of higher order functions, inherent in the closure analysis approach, combined with the complex higher order functions resulting from the encoding of data structures.

Example 2 Both the lower and upper bounds on usage are imprecise in examples such as

```
letrec g w = (hd w) + (tl w)
       f z = (Cons 5 z)
in g (f 3)
```

One would expect `z` to be used at most once, namely in the `(tl w)` summand of the addition, but the analysis finds the usage interval [*Zero*,*Many*]. To see why, consider the translation into core L, annotated with lambda labels:

```
letrec g = λ¹w.(w (λ²n.λ³c.n) (λ⁴x.λ⁵y.x)) + (w (λ⁶n.λ⁷c.n) (λ⁸x.λ⁹y.x))
       f = λ¹⁰z.(((λ¹¹x.λ¹²y.λ¹³n.λ¹⁴c.(c x y)) 5) z)
in g (f 3)
```

The root of the problem is that variable `w` is used twice ($\psi\ 1 = [\mathit{Many},\mathit{Many}]$), and `y` is free in the lambda $\lambda^{13}\ldots$ which is a possible value of `w`. In λ^{13}, variable `y` will be used either [*Zero*,*Zero*] or [*One*,*One*], according as $\mathtt{c}^{14}$ is bound to λ^4 or λ^6. The union of these is [*Zero*,*One*], and the product of this with the usage interval [*Many*,*Many*] of $\mathtt{w}^1$ is [*Zero*,*Many*], which is therefore the usage interval of $\mathtt{y}^{12}$ and hence of $\mathtt{z}^{10}$ in the program.
□

Example 3 Consider the L expression `case` $\mathtt{e}_0$ `of Nil => z (Cons x xs) => z`. One would expect `z` to have lower bound *One* for this expression. However, it is transformed into the core L expression `(e z λx.λxs. z)`. Although closure analysis of $\mathtt{e}_0$ may show that it can evaluate only to $\mathtt{Nil} \equiv \lambda\mathtt{n}.\lambda\mathtt{c}.\mathtt{n}$ and $(\mathtt{Cons}\ \mathtt{e}_1\ \mathtt{e}_2) \equiv \lambda\mathtt{n}.\lambda\mathtt{c}.(\mathtt{c}\ \mathtt{e}_1\ \mathtt{e}_2)$, the $\mathcal{U}_e$ analysis will analyse the two expressions `z` and `λx.λxs.z` separately and find that `z` has usage interval [*Zero*,*One*] in both. The resulting usage interval for `z` is [*Zero*,*Many*], which is informationless. □

This particular problem was caused by the "independent" handling of the two branches of the `case`. Had we instead analysed the expression twice, first with the hypothesis that $\mathtt{e}_0$ evaluates to `Nil`, then with the hypothesis that it evaluates to `(Cons ...)`, we *would* have found the correct usage interval [*One*,*One*] for `z`. However, that would increase the complexity of the analysis considerably and would not solve the problem in Example 2.

5.4.3 Assessment

The usage interval analysis frequently fails to find good lower and upper bounds for expressions involving data structures. The analysis can be improved in various ways, especially to obtain better lower bounds. However, to obtain a reasonably precise yet inexpensive analysis, it is probably necessary to exploit the recursive type structure of data structures, and use the fact that well-structured typed programs tend to handle all the recursive levels the same way.

Type information about recursive data types will be exploited in Chapter 6 below. In particular we shall outline how to do a usage interval analysis for typed first order languages with data structures.

Chapter 6

Evaluation Order Analysis

This chapter describes an analysis to obtain approximate information about subexpression evaluation order. The language considered has lazy data structures and is a typed, lazy first order functional language. The information can be used for optimizing suspensions (or "thunks"), by exploiting knowledge of the form: "this variable definitely has not been evaluated before", or "this variable definitely has been evaluated before". The chapter is based in part on a joint paper with Carsten K. Gomard [30], and much of the text is likely to appear also in his forthcoming Ph.D. thesis [29].

The plan of the chapter is as follows. Section 6.1 introduces backwards path analysis and presents a first order example language with lazy data structures. Section 6.2 introduces variable paths as tools for describing evaluation order of expressions and data structures. Section 6.3 shows how to do evaluation order analysis and presents the main analysis functions. Section 6.4 presents several small examples of evaluation order analysis. Section 6.5 describes the application of evaluation order information to optimization of suspensions. Section 6.6 presents occurrence path analysis, intended as a step towards the analysis based on evaluation order relations presented in Section 6.7. That analysis can be seen as a cheaper approximation to the variable path analysis. Another cheap approximation is the backwards strictness analysis presented in Section 6.8. After this, Section 6.9 shows the extension of evaluation order types to general data types, and Section 6.10 discusses related work.

6.1 Introduction

Previous work on analysing the evaluation order of lazy (or, call by need) languages has focussed on the evaluation order of variables and has not dealt with lazy data structures. Thus Bloss and Hudak would analyse an expression such as

```
letrec f(x,y,z) = if x=0 then y else z
in     f(v+5,w,u+w)
```

and find that the variables `v` and `w` may be evaluated in the following orders: First `v`, then `w`; or first `v`, then `u`, then `w`, assuming that "+" evaluates its arguments from left to right [8]. This finding would be expressed by a set of paths, where a *path* is a finite sequence of variable names. The path set for the above example is $\{\langle$v,w$\rangle$,$\langle$v,u,w$\rangle\}$.

Bloss and Hudak's path analysis is a forwards analysis in which one concatenates the paths for argument expressions in the order in which the corresponding variables are used. Variables can occur at most once in a path: in a lazy language the argument expression bound to a variable is evaluated at most once.

The forwards path analysis method does not work for computations involving lazy data structures, however. The reason is that the order of evaluation of `x` and `y` in the expression `e` $\equiv$ `(Pair x y)` depends on the context of the expression, and more specifically, on the order in which the parts of its result are required. We call this the evaluation order type or context of the expression.

Assume again that "`+`" evaluates its arguments from left to right. Now if the expression `e` $\equiv$ `(Pair x y)` occurs in `(f e)` where `f z = (fst z) + (snd z)`, then `x` is evaluated before `y`. On the other hand, if `e` occurs in `(g e)` where `g z = (snd z) + (fst z)`, `y` would be evaluated before `x`. It is also easy to construct contexts that ignore any one (or both) of the variables.

We therefore suggest an analysis in which one works *backwards*: from the demand order on the result of an expression to its subexpressions, and from these to the variables occurring in the expression. The analysis has not been implemented.

6.1.1 Backwards path analysis

We do a backwards path analysis based on *variable paths*. A path π is a sequence of events. In a variable path, an event p is either ε (epsilon) or the name of a variable `x`. A path π describes one evaluation of `e`, and records the order of evaluation of `e`'s subexpressions. The event `x` marks that the expression bound to `x` first reaches weak head normal form (whnf), whereas the special event ε marks that `e` reaches whnf. (In a lazy language, the expression bound to a variable is evaluated only once).

Given a variable path we can thus tell not only the order in which different variables reach whnf, but also whether a variable reaches whnf before the expression. That is, the epsilons allow us to tell whether the expression is strict in a certain variable.

Evaluation order for a data structure is described by a *context*, or *evaluation order type*, which is the type of the data structure together with path sets describing the order of evaluation of its components. For recursively defined types such as `nlist ::= Nil | Cons nat nlist`, only uniform descriptions are allowed: the recursive components (those of type `nlist`) must all have the same description.

Given a context in form of an evaluation order type for the result of an expression, we find the order in which the free variables are evaluated, and their contexts. Similarly, from the evaluation order type of a function application we get evaluation order information about its arguments.

Example 4 Consider the expression `x + y` and assume again that "+" evaluates its arguments from left to right. The only possible path is $\langle$x,y,$\varepsilon\rangle$ which expresses that first `x` is evaluated to whnf, then `y` is, then the expression reaches its whnf. This shows the expression is strict in `x` and `y`.

For the expression

```
case y of
    Nil       => z
    Cons x xs => x + v
```

the possible paths are $\langle$y,z,$\varepsilon\rangle$ and $\langle$y,x,v,$\varepsilon\rangle$, depending on the branch taken in the `case`. (The variable `x` is not relevant outside the `case` branch in which it occurs; but since we shall assume that all bound variables are distinct, so it will cause no confusion).

Consider the expression `(Cons x z)` and assume it occurs in a context which ignores all heads but uses all tails (such as the context of the argument to function `length`). Then the only possible path is $\langle\varepsilon$,z$\rangle$ which shows that the expression reaches whnf (namely, it evaluates to a `Cons` cell, which is a whnf), and sometime later the context requires the value of `z`. Hence `z` appears in the path, but after ε. □

These examples show that when analysing a program we cannot know which branch is taken in a `case`. Therefore we shall consider all the computations possible for any given program, and work with classes of computations instead of single computations. Hence we work with path sets Π instead of single paths. Such a set is intended to be an upper approximation, that is, a superset of the set of paths actually possible for a given expression.

6.1.2 A first order example language

We use a simple example language to illustrate the evaluation order analysis. It is a lazy, simply typed, first order functional language with (directly or indirectly) recursive data types. Standard examples of recursive data types are tuples, lists of naturals, lists of lists of naturals, *etc.*

A program consists of data type definitions, a set of recursive function definitions, and a program body.

program	::=	*typedef*$_1$ ... *typedef*$_n$ *letrec*	Program with types
typedef	::=	t `::=` *summnd*$_1$ | ... | *summnd*$_n$	Data type declaration
summnd	::=	C *texp*$_1$ ... *texp*$_m$	Type summand
texp	::=	`nat`	Natural numbers
		| t	Type name
letrec	::=	`letrec` *def*$_1$ ... *def*$_n$ `in e`	Program
def	::=	`f ::` $t_1 \times \ldots \times t_n \to t$	Function declaration
		`f` $\mathtt{x}_1 \ldots \mathtt{x}_n$ `= e`	and function definition
`e`	::=	`x`	Function parameter
		| `A` $\mathtt{e}_1 \ldots \mathtt{e}_n$	Base function application
		| C $\mathtt{e}_1 \ldots \mathtt{e}_n$	Constructor application
		| `f` $\mathtt{e}_1 \ldots \mathtt{e}_n$	Function application
		| `case` $\mathtt{e}_0$ `of` *branch*$_1$ ... *branch*$_n$	Case on data type
branch	::=	C $\mathtt{x}_1 \ldots \mathtt{x}_m$ `=> e`	Case branch

A type expression is either a *base type* or the name of a *data type*. A data type is a set of *summands*, each of which consists of a *constructor* and a list of type expressions. Note that data types may be recursive. All constructor names must be distinct.

The language is first order: all defined functions `f`, base functions `A`, and constructors C must be fully applied. A `case` expression `case` $\mathtt{e}_0$ `of` $branch_1$ ... $branch_n$ consists of a *root expression* $\mathtt{e}_0$ the type of which must be a data type t, and several *branches*, one for each constructor of t. The *body* of a program or letrec expression `letrec` ... `in` $\mathtt{e}_0$ is the last expression $\mathtt{e}_0$.

The language is simply (monomorphically) typed and all expressions are assumed to be type correct, but we shall not give a formal type system here.

6.1.3 Informal semantics

Expressions are evaluated or reduced towards weak head normal form (whnf) according to these rules:

- A variable `x` is reduced to whnf by reducing the suspension (*i.e.*, closure) bound to the variable and overwriting it with the whnf (if it is not already reduced to whnf), or by returning the whnf (if already reduced).

- A base function application `A` $\mathtt{e}_1$ ... $\mathtt{e}_n$ is reduced to whnf by reducing the (base type) arguments $\mathtt{e}_1$ ... $\mathtt{e}_n$ to whnf from left to right, then applying the (base valued) function denoted by `A`. The whnf of a base type expression is its value.

- A constructor application C $\mathtt{e}_1$ ... $\mathtt{e}_n$ is reduced to whnf by making suspensions for the arguments $\mathtt{e}_1$ ... $\mathtt{e}_n$, and returning a package consisting of the constructor and these closures.

- An application `f` $\mathtt{e}_1$ ... $\mathtt{e}_n$ of a function defined by `f` $\mathtt{x}_1$ ... $\mathtt{x}_n = \mathtt{e}$ is reduced to whnf as follows. First suspensions are made for the arguments $\mathtt{e}_1$ ... $\mathtt{e}_n$. Then these are bound to the formal parameters $\mathtt{x}_1$ ... $\mathtt{x}_n$, and the body `e` of `f` is reduced to whnf.

- A `case` expression `case` $\mathtt{e}_0$ `of` $branch_1$ ... $branch_n$ is reduced to whnf as follows. First $\mathtt{e}_0$ is reduced to a whnf of form (C_i u_1 ... $\mathrm{u}_{c(i)}$), selecting the i'th case branch C_i $\mathtt{x}_{i1}$ ... $\mathtt{x}_{ic(i)}$ `=>` `e`. Then the suspensions u_1 ... $\mathrm{u}_{c(i)}$ are bound to the formal parameters $\mathtt{x}_{i1}$ ... $\mathtt{x}_{ic(i)}$, and the right hand side $\mathtt{e}_i$ is reduced to whnf.

Thus we assume a pure lazy evaluation or reduction strategy. Namely, we assume it is evaluated non-speculatively, and sequentially "on a single processor". The K machine shown in Chapter 3 would be a useful operational model for this language.

The origin of evaluation order of lazy data structures is the printer's demand on the result of the entire program. This determines the evaluation order type of the program's body expression, which in turn determines the evaluation order type and the evaluation order for all subexpressions in the program.

The order in which the printer demands the result of the program can be described generally as *preorder* and *left to right*. The printer first obtains the weak head normal form to know which constructor to print, then recursively evaluates and prints the arguments to that constructor from left to right.

6.2 Describing evaluation order

In any particular evaluation of a lazy program, the subexpressions of the program will be evaluated to *weak head normal form* (abbreviated *whnf*) in some order, ultimately determined by the printer's demand for a result to print. The precise order cannot be inferred unless one has the program's input data available, so that the program can be run. Our goal is to give an *approximate evaluation order description* (eod) for an expression, valid for all possible evaluations of the program, *independently* of its concrete input data. As outlined above, we shall use path sets for eod's.

6.2.1 Variable paths and their operations

A *path* π is a repetition-free sequence $\langle p_1, \ldots, p_k \rangle$ of events. For *variable paths*, an event is ε (epsilon) or a variable `x`. Let D be a set of events. Then we define *paths* and *path sets* over D as follows:

$$\begin{array}{rcll} \pi & \in & \mathrm{Path}(D) & = \{ \langle p_1, \ldots, p_m \rangle \in D^* \mid p_i = p_j \text{ implies } i = j \} \\ \Pi & \in & \mathrm{PathSet}(D) & = \wp(\mathrm{Path}(D)) \end{array}$$

We shall require that every variable path contains (one) ε. Denoting by $Eod(\mathtt{e})$ the set of possible evaluation order descriptions for `e`, we put $Eod(\mathtt{e}) = \mathrm{PathSet}(Vars(\mathtt{e}) \cup \{\varepsilon\})$, where $Vars(\mathtt{e})$ is the set of variables in `e` (free as well as bound).

The operator $+\!\!+ : D^* \times D^* \to D^*$ is ordinary append of sequences, whereas $+ : \mathrm{Path}(D) \times \mathrm{Path}(D) \to \mathrm{Path}(D)$ is concatenation without duplicates. The (ordinary) prefixing operator is :: for which $p :: \pi = \langle p \rangle +\!\!+ \pi$.

For a path $\pi = \pi_1 +\!\!+ \langle \varepsilon \rangle +\!\!+ \pi_2$ containing ε, we call π_1 the *strict part* of π, and π_2 the *non-strict part*. Recall that if π describes the evaluation of expression `e`, then ε marks the event that `e` reaches whnf. Hence the events in π_1 must happen before `e` reaches whnf, so in a *simply strict* context (requiring only the whnf), the expression is strict in the variables occurring in π_1. This justifies the terminology.

Let Q be a subset of D and let $\pi \in \mathrm{Path}(D)$. By the intersection $\pi \cap Q$ we denote the subsequence of π consisting of elements of Q. For $\Pi \in \mathrm{PathSet}(D)$ we let $\Pi \cap Q$ denote the distributed intersection $\{ \pi \cap Q \mid \pi \in \Pi \}$. Similarly, $\Pi \setminus Q$ means $\Pi \cap (D \setminus Q)$.

We equip $\mathrm{PathSet}(D)$ with the subset ordering $\sqsubseteq_P$, where $\Pi_1 \sqsubseteq_P \Pi_2$ if and only if $\Pi_1 \subseteq \Pi_2$. This makes $\mathrm{PathSet}(D)$ a lattice, with join $\sqcup_P$ being set union, meet $\sqcap_P$ being set intersection, least element $\bot_P = \{\}$, and greatest element $\top_P = \mathrm{Path}(D)$. The least element, $\{\}$, would be the result of analysing a "black hole" such as `f x = f x`. Compare this with the singleton path set $\{\langle \varepsilon \rangle\}$, which would be found for the constant function `f x = 4`.

Assuming that D has n non-ε elements, $\mathrm{Path}(D)$ has $|\mathrm{Path}(D)| = \sum_{i=0}^{n} (n!/(n-i)!)(i+1)$ elements, that is, $(n+1)! \le |\mathrm{Path}(D)| \le (n+2)!$. The height of $\mathrm{PathSet}(D)$ is $|\mathrm{Path}(D)|$, and the number of elements of $\mathrm{PathSet}(D)$ is $|\mathrm{PathSet}(D)| = 2^{|\mathrm{Path}(D)|}$, that is, $s^{(n+1)!} \le |\mathrm{PathSet}(D)| \le 2^{(n+2)!}$.

The *interleaving* $\pi_1 \| \pi_2$ of two paths can be defined as follows

$$\begin{array}{lcl} \| : \text{Path}(D) \times \text{Path}(D) \to \text{PathSet}(D) & & \\ \langle\rangle \parallel \pi_2 & = & \{\ \pi_2\ \} \\ (p :: \pi_1) \parallel \pi_2 & = & \{\ \pi_{21} + \langle p \rangle + \pi' \mid \pi_2 = \pi_{21} +\!\!+\ \pi_{22},\ \pi' \in \pi_1 \parallel \pi_{22}\ \} \end{array}$$

Interleaving a path π with a path set Π gives $\pi \parallel \Pi = \bigcup \{\ \pi \parallel \pi' \mid \pi' \in \Pi\ \}$. The interleaving operations will be needed when we define combinations of path sets.

6.2.2 Argument evaluation order

To describe the order in which a function `f` evaluates its arguments we use an *argument evaluation order description (argument eod)*. This is a variable path set Π_a, where

$$\Pi_a \ \in \ AEod(n) \ = \ \text{PathSet}\{\varepsilon, 1, \ldots, n\}$$

Here ε labels the entire function application, and i labels the i'th argument position. The relation is intended to reflect laziness: To relate evaluations of the arguments $\mathtt{e}_i$, it relates *first* uses of the corresponding variables $\mathtt{x}_i$. Thus the argument eod $\{\langle 1,2,\varepsilon\rangle\}$ does not mean that variable $\mathtt{x}_1$ is not used after variable $\mathtt{x}_2$ in the body of `f`; it means that the first use of $\mathtt{x}_1$ precedes the first use of $\mathtt{x}_2$. In a lazy language this means that argument $\mathtt{e}_1$ is evaluated (once) before argument $\mathtt{e}_2$.

For further examples, $\{\langle 2,1,\varepsilon\rangle\}$ would mean that the second argument will be evaluated (to whnf) before the first argument, and $\{\langle \varepsilon,1\rangle\}$ would mean that the function application will evaluate to a whnf before evaluating its first argument. More interestingly, $\{\langle 1,\varepsilon\rangle\}$ would mean that the first argument will evaluate to a whnf before the function application does. In other words, the function needs its first argument: it is *strict*.

The argument eod for a function is computed by first finding the path set for its body `e`, then extracting information about its formal parameters $\mathtt{x}_1 \ \ldots \ \mathtt{x}_n$ from the path set. This is described in Section 6.3.3 below.

6.2.3 Application contexts

To fully describe a function, we must say not only in which order it consumes its n arguments, but also how it consumes each individual argument value. The latter information shall be called the context or evaluation order type of the argument. We will define evaluation order types in the next section, but for now let $Eot(t)$ denote the set of evaluation order types over type t.

We describe a function by an *application context* $\alpha = (\Pi_a, (\tau_1, \ldots, \tau_n))$, *i.e.*, a pair of an argument eod Π_a and a tuple $(\tau_1, \ldots, \tau_n)$ of evaluation order types for the arguments. Let `f` be a function of type $t_1 \times \ldots \times t_n \to t$. The set $Actx(t_1 \ \ldots \ t_m)$ of possible application contexts is defined as follows:

$$\alpha \ \in \ Actx(t_1 \ \ldots \ t_m) \ = \ AEod(m) \times \prod_{j=1}^{m} Eot(t_j)$$

Assuming that $Eot(t_j)$ is a lattice of finite height for each j, $Actx(t_1 \ \ldots \ t_m)$ is also a lattice of finite height (with $\sqsubseteq_P$ taken componentwise).

Note that an application context does *not* describe the order of evaluation between *components* of the values of distinct argument expressions $\mathtt{e}_i$ and $\mathtt{e}_j$; only the order of evaluation to whnf of $\mathtt{e}_i$ and $\mathtt{e}_j$, and the order of evaluation inside each of these.

6.2.4 Evaluation order types

An evaluation order type (eot) describes the evaluation order the components of a value. Evaluation order types over base types such as `nat` are quite trivial, because `nat` values are atomic. Since a base value can be considered a nullary constant function, it is natural to take evaluation order types over base types to be elements of $AEod(0)$. There are only two evaluation order descriptions in $AEod(0) = \{\{\}, \{\langle\varepsilon\rangle\}\}$, representing the contradictory context and the full evaluation context, respectively.

Evaluation order types over values of a data type t are more interesting. We shall first discuss evaluation order types for non-recursive data types, then consider one example of a recursive data type, `nlist ::= Nil | Cons nat nlist`. To avoid excessive technicalities we will consider the general case of recursive types only in Section 6.9.

6.2.4.1 Constructor contexts

Data types involve constructors, so to introduce evaluation order types, one must understand how to represent the evaluation order of constructors' arguments. For an m-ary constructor we will describe the evaluation order of its m arguments by an argument eod $\Pi_a \in AEod(m)$.

Since we also have to include the contexts of the constructor's arguments, we describe a constructor C of type $t_1 \times \ldots \times t_m \to t$ and arity m by an application context in $Actx(t_1 \ldots t_m)$. That is, we describe it in the same way as a function[1]. The only difference is that a constructor application always reaches whnf (namely, it evaluates to the constructor) before its arguments get evaluated (on demand from the context), so its argument eod's will all have form $\{\langle\varepsilon,\ldots\rangle,\ldots,\langle\varepsilon,\ldots\rangle\}$.

6.2.4.2 Non-recursive data types

For a given (non-recursive) data type t, we can finally define an *evaluation order type (eot)* τ to be a tuple $(\alpha_1,\ldots,\alpha_m)$ of application contexts, one for each constructor. We shall also use the term *context* to mean eot. The set $Eot(t)$ of evaluation order types can now be defined as follows (for non-recursive types):

$$\begin{array}{rcl} Eot(\mathtt{nat}) & = & AEod(0) \\ \tau \in Eot(t) & = & \prod_{i=1}^{n} Actx(t_{i1} \ldots t_{ic(i)}) \\ & & \text{where } t ::= C_1\ t_{11}\ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1}\ldots t_{nc(n)} \end{array}$$

For every non-recursive data type t it is possible to unfold all the occurrences of $Actx$ and (recursively) Eot, ending up with a finite product of $AEod$'s. Thus the ordering inherited

[1] In fact, the usual non-recursive encoding of constructors as higher order functions shows that the possible constructor contexts can be construed as functions: the functions which consume the value (see Section 3.3).

from the component lattices $AEod(m)$ makes the set $Eot(t)$ into a complete lattice of finite height. Its least element is denoted by $\bot_{Eot(t)}$.

Clearly this definition of $Eot(t)$ will not work for directly or indirectly recursively defined data types; it would lead to an infinite product.

6.2.4.3 Recursive data types

For a recursive data type such as `nlist ::= Nil | Cons nat nlist`, it is somewhat more complicated to describe the evaluation order types. There are several ways to solve this problem, but they are either quite conservative or lead to notational problems when stated in full generality. We shall therefore postpone the general treatment of recursive data types to Section 6.9 below.

Here we just postulate the existence of two operations on eots. The *unfolding* $\alpha_i = \tau @ C_i$ extracts C_i's application context α_i from τ, and the *folding* $t[\alpha_1,\ldots,\alpha_n]$ constructs an eot over type t from application contexts $\alpha_1, \ldots, \alpha_n$.

We focus now on the special case of `nlist ::= Nil | Cons nat nlist`. The key to a solution is to identify the recursive occurrence of `nlist` with the top level one. This means that evaluation order information for all the elements of a list are identified; the information is required to be *uniform* over the list elements. As a consequence, the analysis can collect only information which is valid for all elements of a list. This is not so bad as it may seem: well-structured typed functional programs usually treat all elements of recursive data types the same anyway.

Following the general schema for non-recursive types, but ignoring the recursive structure, we obtain

$$\begin{aligned} Eot(\texttt{nlist}) &= Actx() \times Actx(\texttt{nat nlist}) \\ &= AEod(0) \times AEod(2) \times AEod(0) \end{aligned}$$

Here the first $AEod(0)$ describes the context of `Nil`, $AEod(2)$ describes the context of `Cons`, and the second $AEod(0)$ describes the context of the `nat` argument to `Cons`. In examples we will usually ignore the information about `Nil` and `nat`, and represent $Eot\{\texttt{nlist}\}$ by $AEod(2)$ only. In those cases we use the notation $\texttt{nlist}_{\texttt{Cons}:\Pi}$ for the evaluation order type over `nlist` corresponding to the path set $\Pi \in \mathrm{PathSet}(\{\varepsilon,1,2\})$ for `Cons`.

Let us consider the intuitive interpretation of the evaluation order types over `nlist`. For example, $\texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1\rangle,\langle\varepsilon,1,2\rangle\}}$ is the context induced by the printer, which is strict in the head and evaluates the head before the tail. The context of the argument to the function `reverse` which reverses a list of `nat`'s (see Section 6.4) is $\texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}}$ when the result is in context $\texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}}$. Similarly, $\texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2\rangle\}}$ is a tail strict context which will ignore all heads (*i.e.*, elements) of the list; this is the context of the argument to function `length`. More examples of evaluation order types over `nlist` can be found in the analysis examples given in Section 6.4 below.

6.3 Evaluation order analysis

The point of the evaluation order analysis is to find evaluation order information about a given program `letrec ... in` $\mathtt{e}_0$. We shall represent this information as an *application context description* ζ such that for every function `f` of type $t_1 \times \ldots \times t_n \to t$ in the program,

$$\zeta\mathtt{f} \;:\; \mathit{Eot}(t) \to \mathit{Actx}(t_1 \ldots t_n)$$

The idea is that $\zeta\mathtt{f}$ maps the context τ of an application $(\mathtt{f}\ \mathtt{e}_1 \ldots \mathtt{e}_n)$ of `f` into an application context $(\Pi_a, (\tau_1, \ldots, \tau_n))$, which consists of an argument eod (a path set) Π_a, and an evaluation order type τ_i for each argument expression $\mathtt{e}_i$. The set of ζ's for a given program is called ADescription for application context description.

Below we present two analysis functions, $\mathcal{R}$ and $\mathcal{T}$, which are used when computing the description ζ. These functions in turn rely on ζ. Therefore ζ will later be defined as a solution to recursive equations involving these analysis functions.

6.3.1 Combining variable path sets

To obtain a compositional evaluation order analysis, we need a way to combine paths for subexpressions into paths for the containing expression.

Consider an expression $\mathtt{e} \equiv (\mathtt{op}\ \mathtt{e}_1\ \mathtt{e}_2)$ with subexpressions $\mathtt{e}_1$ and $\mathtt{e}_2$, and assume that we have a path π_i for each subexpression $\mathtt{e}_i$. What can we say about paths for the whole of `e`?

Assume that $(\mathtt{op}\ \mathtt{e}_1\ \mathtt{e}_2)$ first evaluates $\mathtt{e}_1$ to whnf, then $\mathtt{e}_2$, then reaches a whnf itself. Then we know that the strict part of π_1 (*i.e.*, the part preceding ε) must precede all of π_2, and that the strict part of π_2 must precede ε in any path for `e`. This is because the strict part of π_1 is known to happen before $\mathtt{e}_1$ reaches whnf, and we assumed that this happens before $\mathtt{e}_2$ gets evaluated.

On the other hand, we know nothing about the relative order of the non-strict parts of π_1 and π_2, or whether they come before or after ε in the resulting path. Consequently we must consider all interleavings of the non-strict parts of π_1, π_2, and $\langle\varepsilon\rangle$.

The assumption about $(\mathtt{op}\ \mathtt{e}_1\ \mathtt{e}_2)$ could be given as a path $\pi_a = \langle 1, 2, \varepsilon\rangle$, so the possible paths for `e` can be given as a formal combination of paths or path sets, and the preceding discussion (sort of) motivates the following definition.

Definition 1 Let $\Pi_a \in \mathrm{PathSet}\{\varepsilon, 1, \ldots, n\}$ be a path set, called the *combiner*, and let $\Pi_i \in \mathrm{PathSet}(D_i)$, $i = 1, \ldots, n$. Then $\Pi = \Pi_a(\Pi_1,\ldots,\Pi_n)$ is a path set in $\mathrm{PathSet}(\bigcup_{i=1}^{n} D_i)$, called the Π_a *combination* of $\Pi_1 \ldots \Pi_n$. It is defined as follows:

$$\Pi \;=\; \bigcup \{\ \pi_a(\pi_1,\ldots,\pi_n) \mid \pi_a \in \Pi_a,\ \pi_i \in \Pi_i\ \}$$

where the single path combination $\pi_a(\pi_1,\ldots,\pi_n)$ is defined by

$$\begin{array}{lcl}
\langle\rangle(\pi_1,\ldots,\pi_n) & = & \{\ \langle\rangle\ \} \\
(\varepsilon{::}\pi)(\pi_1,\ldots,\pi_n) & = & \{\ \varepsilon :: \pi' \mid \pi' \in \pi(\pi_1,\ldots,\pi_n)\ \} \\
(i{::}\pi)(\pi_1,\ldots,\pi_n) & = & \{\ \pi_{i1} {+} \pi' \mid \pi_i {=} \pi_{i1} {+\!\!+} \langle\varepsilon\rangle {+\!\!+} \pi_{i2} \text{ and } \pi' {\in} \pi_{i2} \| \pi(\pi_1,\ldots,\pi_n)\ \}
\end{array}$$

Path interleaving $\|$ was defined in Section 6.2.1. As noted above, in the combination $(i :: \pi)(\pi_1,\ldots,\pi_n)$ only the strict part π_{i1} (before ε) of π_i is guaranteed to precede the rest $\pi(\pi_1,\ldots,\pi_n)$ of the combination. For the non-strict part π_{i2} we must consider all interleavings with the rest. □

6.3.2 Expression evaluation order in the example language

For each syntactic construct in the example language we show how to put together path sets of subexpressions to obtain a path set for the entire construct. This will be used in analysis function $\mathcal{R}$. For constructor and function application we can rely on the combination operation defined above, but for variables, base function application, and `case` we define "plumbing" operations R_{var}, R_{An}, and $R_{\texttt{case},n}$ to take care of this. These reflect the informal evaluation rules we gave in Section 6.1.3.

Let Π_i denote the path set of subexpression $\texttt{e}_i$, $i = 0, \ldots, n$, and let Π denote the desired path set for `e`.

- $\texttt{e} \equiv \texttt{x}$:

  Variable `x` must reach whnf for `e` to reach whnf; hence the only possible path set is $\Pi = R_{var}(\texttt{x}) = \{\langle \texttt{x},\varepsilon\rangle\}$.

- $\texttt{e} \equiv (\texttt{A e}_1 \; \ldots \; \texttt{e}_n)$:

  Base functions are strict and evaluate their *base type* arguments from left to right, and evaluation to whnf is complete evaluation. The combined path set is therefore simply $\Pi = R_{An}\Pi_1, \ldots, \Pi_n = (\Pi_1 \backslash \{\varepsilon\}) + \ldots + (\Pi_n \backslash \{\varepsilon\}) + \langle\varepsilon\rangle$.

- $\texttt{e} \equiv (C_i \; \texttt{e}_1 \; \ldots \; \texttt{e}_m)$:

  The application context for C_i in the context τ for `e` is $\alpha = \tau @ C_i$, which has form $(\Pi_a, (\tau_1, \ldots, \tau_m))$. The path set for `e` is the variable path combination $\Pi = \Pi_a(\Pi_1,\ldots,\Pi_m)$.

- $\texttt{e} \equiv (\texttt{f e}_1 \; \ldots \; \texttt{e}_n)$:

  The function description $\zeta \texttt{f} \tau$ gives an application context α of form $(\Pi_a, (\tau_1, \ldots, \tau_n))$. The path set Π for `e` is computed as the variable path combination $\Pi_a(\Pi_1,\ldots,\Pi_n)$.

- $\texttt{e} \equiv \begin{pmatrix} \texttt{case } \texttt{e}_0 \texttt{ of} \\ \quad C_1 \; \texttt{x}_{11}\ldots\texttt{x}_{1c(1)} \texttt{=>e}_1 \\ \quad \ldots \\ \quad C_n \; \texttt{x}_{n1}\ldots\texttt{x}_{nc(n)} \texttt{=>e}_n \end{pmatrix}$:

  The root expression $\texttt{e}_0$ is evaluated to whnf first, then *one* of the branches is evaluated. This combination is conveniently described as a variable path combination, where the combiner is $R_{\texttt{case},n} = \{ \langle 0, 1, \varepsilon\rangle, \langle 0, 2, \varepsilon\rangle, \ldots, \langle 0, n, \varepsilon\rangle \}$.

Note that function application is the only case in which the combination is not determined by the syntactic construct and its context. Hence the need for an application context description ζ in the analysis.

6.3.3 Argument evaluation order

Consider a function definition `f x`$_1$ `x`$_2$ `= e`, and let $\pi = \langle \mathtt{y},\mathtt{x}_2,\varepsilon,\mathtt{x}_1,\mathtt{z}\rangle$ be a path for evaluation of `e`. To obtain an argument eod for `f`, we extract the subpath $\langle \mathtt{x}_2,\varepsilon,\mathtt{x}_1\rangle$ of π, then we construct the index path $\langle 2, \varepsilon, 1\rangle$. For a `case` branch C `x`$_1$ `x`$_2$ `=> e` the procedure is the same, except that all index paths begin with epsilons.

This extraction of argument eod's from the path set of an expression is done by the functions $\mathcal{A}_\varepsilon$ for defined functions and by $\mathcal{A}$ for constructors.

$$\mathcal{A},\, \mathcal{A}_\varepsilon : \mathrm{PathSet}(D) \times \mathrm{Var}^n \rightarrow \mathrm{PathSet}\{1,\ldots,n,\varepsilon\}$$

$$\mathcal{A}_\varepsilon(\Pi,(\mathtt{x}_1,\ldots,\mathtt{x}_n)) = \{\ \mathit{varindex}(\pi) \mid \pi \in \Pi \cap \{\varepsilon,\mathtt{x}_1,\ldots,\mathtt{x}_n\}\}$$

$$\mathcal{A}(\Pi,(\mathtt{x}_1,\ldots,\mathtt{x}_n)) = \{\ \varepsilon :: \mathit{varindex}(\pi) \mid \pi \in \Pi \cap \{\mathtt{x}_1,\ldots,\mathtt{x}_n\}\ \}$$

Function *varindex* computes a path of variable indices i (or ε) describing the order of occurrence of $\mathtt{x}_1,\ldots,\mathtt{x}_n$ in a variable path:

$$\mathit{varindex}\text{: } \mathrm{Path}(D) \rightarrow \mathrm{Path}\{\varepsilon,1,\ldots,n\}$$

$$\begin{aligned}
\mathit{varindex}(\langle\rangle) &= \langle\rangle \\
\mathit{varindex}(\varepsilon :: \pi) &= \langle\varepsilon\rangle + \mathit{varindex}(\pi) \\
\mathit{varindex}(\mathtt{x}_i :: \pi) &= \langle i\rangle + \mathit{varindex}(\pi)
\end{aligned}$$

6.3.4 Evaluation order analysis functions

The *evaluation order analysis function* $\mathcal{R}$ computes an evaluation order description (a variable path set) for a given expression `e` of type t in given context τ. All variables (both formal parameters of functions and pattern variables in `case`) must be distinct.

$$\mathcal{R}[\![\,\mathtt{e}\,]\!] : \mathrm{ADescription} \rightarrow \mathit{Eot}(t) \rightarrow \mathit{Eod}(\mathtt{e})$$

$$\begin{aligned}
\mathcal{R}[\![\,\mathtt{x}\,]\!]\zeta\tau &= R_{var}(\mathrm{x}) \\
\mathcal{R}[\![\,\mathtt{A}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n\,]\!]\zeta\tau &= R_{An}(\mathcal{R}[\![\,\mathtt{e}_1\,]\!]\zeta\tau,\ \ldots,\ \mathcal{R}[\![\,\mathtt{e}_n\,]\!]\zeta\tau) \\
\mathcal{R}[\![\,C_i\ \mathtt{e}_1\ \ldots\ \mathtt{e}_m\,]\!]\zeta\tau\mathtt{y} &= a(\mathcal{R}[\![\,\mathtt{e}_1\,]\!]\zeta\tau_1,\ \ldots,\ \mathcal{R}[\![\,\mathtt{e}_m\,]\!]\zeta\tau_m) \\
&\ \text{where } (a,(\tau_1,\ldots,\tau_m)) = \tau @ C_i \\
\mathcal{R}[\![\,\mathtt{f}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n\,]\!]\zeta\tau &= a(\mathcal{R}[\![\,\mathtt{e}_1\,]\!]\zeta\tau_1,\ \ldots,\ \mathcal{R}[\![\,\mathtt{e}_n\,]\!]\zeta\tau_n) \\
&\ \text{where } (a,(\tau_1,\ldots,\tau_n)) = \zeta\mathtt{f}\tau
\end{aligned}$$

$$\mathcal{R}\left[\!\!\left[\begin{array}{l}\mathtt{case}\ \ \mathtt{e}_0\ \ \mathtt{of} \\ \qquad C_1\ \ \mathtt{x}_{11}\ldots\mathtt{x}_{1c(1)}\ \mathtt{=>e}_1 \\ \qquad \ldots \\ \qquad C_n\ \ \mathtt{x}_{n1}\ldots\mathtt{x}_{nc(n)}\mathtt{=>e}_n\end{array}\right]\!\!\right]\zeta\tau = R_{\mathtt{case},n}(r_0,r_1,\ldots,r_n)$$

$$\begin{aligned}
\text{where } r_0 &= \mathcal{R}[\![\,\mathtt{e}_0\,]\!]\zeta\tau_0 \\
\tau_0 &= t_0[\alpha_1,\ldots,\alpha_n] \\
\alpha_i &= (a_i,\ (\tau_{i1},\ldots,\tau_{ic(i)})) \\
a_i &= \mathcal{A}(r_i,(\mathtt{x}_{i1},\ldots,\mathtt{x}_{ic(i)})) \\
\tau_{ij} &= \mathcal{T}[\![\,\mathtt{e}_i\,]\!]\zeta\tau\mathtt{x}_{ij} \\
r_i &= \mathcal{R}[\![\,\mathtt{e}_i\,]\!]\zeta\tau \\
i \in \{1,\ldots,n\},&\ \ j \in \{1,\ldots,c(i)\}
\end{aligned}$$

The first rules should be self-explanatory after Section 6.3.2 above.

For an application of constructor C_i, the pertinent application context $(a,(\tau_1,\ldots,\tau_m)) = \tau@C_i$ is extracted from the context τ. The argument expressions $\mathtt{e}_i$ are analysed in their contexts τ_i, and the path sets combined using the combiner a. (Notation: We use a and r instead of Π_a and Π for generality; in later sections the a will denote other things than path sets).

For a function application, the application context $(a,(\tau_1,\ldots,\tau_m))$ must first be found using ζ, then the argument expressions are analysed in these contexts, and the path sets combined using the combiner a.

To find the evaluation order on a `case` expression, we must analyse $\mathtt{e}_0$ in the context τ_0 found by analysing the branches with $\mathcal{R}$, and we must analyse the branches in the same context as the entire `case` expression. To find the uniform context τ_0 of $\mathtt{e}_0$, we must combine the application contexts α_i for each of the constructors C_i using the operation $t_0[\alpha_1,\ldots,\alpha_n]$, where t_0 is the type of $\mathtt{e}_0$. The application contexts are found by computing the context τ_{ij} of each pattern variable $\mathtt{x}_{ij}$ recursively using $\mathcal{T}$, and by finding the argument eod a_i for each constructor. Finding a_i in turn requires finding the evaluation order r_i on the right hand side $\mathtt{e}_i$. Finally we can combine the evaluation order descriptions $r_0, r_1, \ldots, r_n$ using the combiner $R_{\mathtt{case},n}$.

The *context analysis function* $\mathcal{T}$ computes the context (evaluation order type) of a variable `y` in a given expression `e` which is in context τ. The context analysis function is defined as follows, where t is the type of expression `e`, and $\mathrm{Var}(t_1)$ is the set of variables of type t_1.

$$\mathcal{T}[\![\,\mathtt{e}\,]\!] : \text{ADescription} \to \mathit{Eot}(t) \to \mathrm{Var}(t_1) \to \mathit{Eot}(t_1)$$

$$\begin{array}{lll}
\mathcal{T}[\![\,\mathtt{x}\,]\!]\zeta\tau\mathtt{y} & = \tau & \text{if } \mathtt{x} \equiv \mathtt{y} \\
 & = \bot_{\mathit{Eot}(t_1)} & \text{otherwise} \\
\mathcal{T}[\![\,\mathtt{A}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n\,]\!]\zeta\tau\mathtt{y} & = \bigsqcup_{i=1}^{n} \mathcal{T}[\![\,\mathtt{e}_i\,]\!]\zeta\tau\mathtt{y} & \\
\mathcal{T}[\![\,C_i\ \mathtt{e}_1\ \ldots\ \mathtt{e}_m\,]\!]\zeta\tau\mathtt{y} & = \bigsqcup_{j=1}^{m} \mathcal{T}[\![\,\mathtt{e}_j\,]\!]\zeta\tau_j\mathtt{y} & \\
 & \quad \text{where } (a,(\tau_1,\ldots,\tau_m)) = \tau@C_i & \\
\mathcal{T}[\![\,\mathtt{f}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n\,]\!]\zeta\tau\mathtt{y} & = \bigsqcup_{i=1}^{n} \mathcal{T}[\![\,\mathtt{e}_i\,]\!]\zeta\tau_i\mathtt{y} & \\
 & \quad \text{where } (a,(\tau_1,\ldots,\tau_n)) = \zeta\mathtt{f}\tau &
\end{array}$$

$$\mathcal{T}\left[\!\!\left[\begin{array}{l}\mathtt{case}\ \ \mathtt{e}_0\ \ \mathtt{of} \\ \qquad C_1\ \ \mathtt{x}_{11}\ldots\mathtt{x}_{1c(1)}\ \mathtt{=>}\mathtt{e}_1 \\ \qquad \ldots \\ \qquad C_n\ \ \mathtt{x}_{n1}\ldots\mathtt{x}_{nc(n)}\mathtt{=>}\mathtt{e}_n\end{array}\right]\!\!\right]\zeta\tau\mathtt{y} = \mathcal{T}[\![\,\mathtt{e}_0\,]\!]\zeta\tau_0\mathtt{y} \sqcup \left(\bigsqcup_{i=1}^{n} \mathcal{T}[\![\,\mathtt{e}_i\,]\!]\zeta\tau\mathtt{y}\right)$$

$$\begin{array}{lrcl}
\text{where} & \tau_0 & = & t_0[\alpha_1,\ldots,\alpha_n] \\
 & \alpha_i & = & (a_i,\ (\tau_{i1},\ldots,\tau_{ic(i)})) \\
 & a_i & = & \mathcal{A}(r_i,(\mathtt{x}_{i1},\ldots,\mathtt{x}_{ic(i)})) \\
 & \tau_{ij} & = & \mathcal{T}[\![\,\mathtt{e}_i\,]\!]\zeta\tau\mathtt{x}_{ij} \\
 & r_i & = & \mathcal{R}[\![\,\mathtt{e}_i\,]\!]\zeta\tau \\
 & \multicolumn{3}{l}{i \in \{1,\ldots,n\}} \\
 & \multicolumn{3}{l}{j \in \{1,\ldots,c(i)\}}
\end{array}$$

If `x` and `y` are the same variable, then an occurrence of `x` contributes to the context of `y`, otherwise it does not (*i.e.*, it contributes with the least context $\bot_{Eot(t_1)}$ of variable `y`).

The `case` equation is very similar to that in function $\mathcal{R}$ above, and the explanation is the same. Recall that all variables are distinct, so there is no confusion possible between `y` and the pattern variables $\mathtt{x}_{ij}$ in `case`.

6.3.5 Doing evaluation order analysis

The analysis result we seek for a given program is the least solution $\zeta \in$ ADescription to the simultaneous equations

$$\begin{aligned} \zeta\mathtt{f}\tau \;=\; & (a, (\tau_1, \ldots, \tau_n)) \\ & \text{where } \begin{aligned}[t] a &= \mathcal{A}_\varepsilon(r,(\mathtt{x}_1,\ldots,\mathtt{x}_n)) \\ \tau_i &= \mathcal{T}[\![\,\mathtt{e}\,]\!]\zeta\tau\mathtt{x}_i \qquad \text{for } i = 1,\ldots,n \\ r &= \mathcal{R}[\![\,\mathtt{e}\,]\!]\zeta\tau \end{aligned} \\ & \text{and} \quad \mathtt{f}\ \mathtt{x}_1\ldots\mathtt{x}_n\ \mathtt{=}\ \mathtt{e} \text{ in the program has type } \mathtt{f} : t_1\times\ \ldots\ \times t_n \to t \end{aligned}$$

This least solution exists and can be found in finite time, because ADescription is a lattice of finite height and the analysis functions $\mathcal{A}_\varepsilon$, $\mathcal{T}$, and $\mathcal{R}$ are all monotonic in their arguments r, ζ, and τ. (ADescription has finite height by virtue of $AEod(n)$ and $Eot(t)$ being lattices of finite height, and the program being finite).

To analyse a given program `letrec ... in` $\mathtt{e}_0$ where the goal expression $\mathtt{e}_0$ has type t_0, we start by finding the evaluation order type τ_0 expressing the printer's order of demand on values of type t_0. If t_0 is a base type, then this evaluation order type is the simple demand $\{\langle\varepsilon\rangle\}$. If t_0 is `nlist`, then τ_0 can be represented as the path set $\Pi_a = \{\langle\varepsilon,1\rangle,\langle\varepsilon,1,2\rangle\}$, cf. Section 6.2.4.3. For other data types, τ_0 can be expressed similarly, reflecting that the printer evaluates and prints all data structures in preorder and from left to right.

When we have found τ_0, we analyse the goal expression $\mathtt{e}_0$ in that context and find all the possible contexts for applications of defined function `f`. The least upper bound $\tau_\mathtt{f}$ of these contexts represents evaluation order information which is valid for all evaluations of the program. The same therefore is the case for the eod (a variable path set) $\Pi_\mathtt{f} = \mathcal{R}[\![\,\mathtt{e}\,]\!]\zeta\tau_\mathtt{f}$ for the body `e` of `f` computed from $\tau_\mathtt{f}$.

Thus the information in $\Pi_\mathtt{f}$ can validly be used to optimize the body of `f`.

6.3.6 Complexity of variable path analysis

Here we shall give very rough estimates of the complexity of variable path analysis by studying the size of the lattice containing $\zeta\mathtt{f}$ as a function of the type of `f`. This gives an upper bound on the number of iterations needed to compute the fixed point ζ. Since the analysis is backwards, an abstract function $\zeta\mathtt{f}$ maps a context (an evaluation order type) for the function's result type into an application context for the function. Hence the function's result type will influence the complexity more than its argument types.

Using the facts about the height and number of elements of PathSet(D) found in Section 6.2.1 above, we see that the height of the argument eod lattice $AEod(n)$ is between

$(n+1)!$ and $(n+2)!$, and its number of elements is between $2^{(n+1)!}$ and $2^{(n+2)!}$. This is the number of possible argument eod's for an n-argument function. More precisely, the height of $AEod(0)$ is 1, the height of $AEod(1)$ is 3, the height of $AEod(2)$ is 11, the height of $AEod(4)$ is 49, the height of $AEod(5)$ is 261, *etc.* The number of elements is 2 to the power of the height in every case.

When argument eod's are used for describing constructors, we require that ε is the first element of each path, so there are not so many argument eod's. For an m-argument constructor the number of different paths is $\sum_{i=0}^{n}(n!/(n-i)!)$, which is also the height of the argument eod lattice, and is between $n!$ and $(n+1)!$.

More precisely, considering only constructor descriptions, the height of $AEod(0)$ is 1, the height of $AEod(1)$ is 2, the height of $AEod(2)$ is 5, the height of $AEod(4)$ is 16, the height of $AEod(5)$ is 65, *etc.* The number of elements is 2 to the power of the height in every case.

Specifically, the height of $Eot(\texttt{nat}) = AEod(0)$ is 1 and the number of elements is 2. The height of $Eot(\texttt{nlist}) = \mathrm{AEod}(0) \times AEod(2) \times AEod(0)$ is $1+5+1=7$, and the number of elements is $2^7 = 128$. Hence for the function `take :: nlist` $\times$ `nat` $\rightarrow$ `nlist`, the lattice of abstract functions $\zeta(\texttt{take})$ is $Eot(\texttt{nlist}) \rightarrow AEod(2) \times (Eot(\texttt{nlist}) \times Eot(\texttt{nat}))$, which has height $128*(11+(7+1)) = 2432$, which is rather large. Even worse, the height of the lattice of abstract functions grows very fast with the complexity of the result type. It is at least $2^{m!}$ where m is the largest constructor arity in the result type.

In later sections we shall see that the evaluation order analysis functions $\mathcal{R}$ and $\mathcal{T}$ defined in this section can support a whole range of analyses, some of which are less expensive than the variable path analysis. The analysis functions $\mathcal{R}$ and $\mathcal{T}$ are parametrized on the following items: The lattices $Eod(\texttt{e})$ and $AEod(n)$ with the least upper bound operation $\sqcup$, and the functions R_{var}, R_{An}, and $R_{\texttt{case},n}$, the combination operation $a(r_1,\ldots,r_n)$, and the functions $\mathcal{A}$ and $\mathcal{A}_\varepsilon$ for extracting argument eod's from eod's.

6.4 Examples of variable path analysis

In this section we will show the results of applying the variable path analysis to some small programs. We shall always assume that the function is always at least in a simply strict context. The result of analysing a function `f` of type $t_1 \times \ldots \times t_n \rightarrow t$ will be shown in the following form:

$$\texttt{f} : \Pi_a,\ \tau_1 \times \ldots \times \tau_n \rightarrow \tau$$

meaning that $\zeta \texttt{f} \tau = (\Pi_a, (\tau_1 \ldots \tau_n))$. We write evaluation order types τ as the type name with path sets for the constructors subscripted.

The function length

```
nlist ::= Nil | Cons nat nlist
length :: nlist → nat
length xs = case xs of
              Nil       => 0
              Cons y ys => 1 + (length ys)
```

Analysis result:

$$\texttt{length} : \{\langle 1,\varepsilon\rangle\}, \texttt{nlist}_{\text{Cons}:\{\langle\varepsilon,2\rangle\}} \rightarrow \texttt{nat}$$

The argument path set $\{\langle 1,\varepsilon\rangle\}$ shows that `length` is strict, and the context for `xs`, $\texttt{nlist}_{\text{Cons}:\{\langle\varepsilon,2\rangle\}}$, means that all list elements (heads) are ignored and that the function is tail strict.

The function sum

```
sum :: nlist → nat
sum xs = case xs of
           Nil       => 0
           Cons y ys => y + (sum ys)
```

Analysis result:

$$\texttt{sum} : \{\langle 1,\varepsilon\rangle\}, \texttt{nlist}_{\text{Cons}:\{\langle\varepsilon,1,2\rangle\}} \rightarrow \texttt{nat}$$

The `sum` function is strict, evaluates head before tail, and is head and tail strict.

The function and

```
bool     ::= False | True
boollist ::= Nil | Cons bool boollist
and :: boollist → bool
and xs = case xs of
           Nil       => True
           Cons y ys => case y of
                          False => False
                          True  => and ys
```

Analysis result:

$$\texttt{and} : \{\langle 1,\varepsilon\rangle\}, \texttt{boollist}_{\text{Cons}:\{\langle\varepsilon,1\rangle,\langle\varepsilon,1,2\rangle\}} \rightarrow \texttt{bool}$$

The `and` function is strict, evaluates head before tail, and is a natural example of a function which is head strict but not tail strict.

One version of the function take

```
take :: nlist × nat → nlist
take xs n = case xs of
              Nil       => Nil
              Cons y ys => case n=0 of
                             True  => Nil
                             False => Cons y (take ys (n-1))
```

Analysis results:

$$\texttt{take} : \{\langle 1,\varepsilon\rangle,\langle 1,2,\varepsilon\rangle\},\ \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon\rangle,\langle\varepsilon,1,2\rangle\}} \times \texttt{nat} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}}$$

$$\texttt{take} : \{\langle 1,\varepsilon\rangle,\langle 1,2,\varepsilon\rangle\},\ \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon\rangle,\langle\varepsilon,2,1\rangle\}} \times \texttt{nat} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}}$$

That `take` is strict in `xs` and evaluates `xs` before `n` is independent of the context. Even in a head and tail strict context this version of `take` is neither head nor tail strict in `xs`. In an $\texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}}$ context, `xs` is in an $\texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon\rangle,\langle\varepsilon,1,2\rangle\}}$ context, and in an $\texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}}$ context, `xs` is in context $\texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon\rangle,\langle\varepsilon,2,1\rangle\}}$.

Another version of the function take

```
take :: nlist × nat → nlist
take xs n = case n=0 of
             True  => Nil
             False => case xs of
                        Nil       => Nil
                        Cons y ys => Cons y (take ys (n-1))
```

Analysis results:

$$\texttt{take} : \{\langle 2,\varepsilon\rangle,\langle 2,1,\varepsilon\rangle\},\ \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1\rangle,\langle\varepsilon,1,2\rangle\}} \times \texttt{nat} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}}$$

$$\texttt{take} : \{\langle 2,\varepsilon\rangle,\langle 2,1,\varepsilon\rangle\},\ \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1\rangle,\langle\varepsilon,2,1\rangle\}} \times \texttt{nat} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}}$$

This version of `take` is strict in `n` and evaluates `n` before `xs` independently of the context. Like the previous version it is not tail strict `xs` in a tail strict context, but it is head strict in a head strict context. Also, it propagates the order of evaluation from the context to the argument as does the previous version.

Reverse function with accumulating parameter

```
reverse :: nlist × nlist → nlist
reverse xs zs = case xs of
                 Nil       => zs
                 Cons y ys => reverse ys (Cons y zs)
```

Analysis results:

$$\begin{aligned}
\texttt{reverse} : {} & \{\langle 1,\varepsilon\rangle,\langle 1,2,\varepsilon\rangle\},\\
& \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2\rangle\}} \times \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon\rangle\}}\\
\texttt{reverse} : {} & \{\langle 1,\varepsilon\rangle,\langle 1,2,\varepsilon\rangle\},\\
& \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2\rangle,\langle\varepsilon,2,1\rangle\}} \times \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1\rangle,\langle\varepsilon,1,2\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1\rangle,\langle\varepsilon,1,2\rangle\}}\\
\texttt{reverse} : {} & \{\langle 1,2,\varepsilon\rangle,\langle 1,\varepsilon,2\rangle\},\\
& \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}} \times \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}}\\
\texttt{reverse} : {} & \{\langle 1,2,\varepsilon\rangle,\langle 1,\varepsilon,2\rangle\},\\
& \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}} \times \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}}
\end{aligned}$$

Even in a context which only demands a whnf (first line), `reverse` is strict in `xs`, evaluates it before `zs`, and is tail strict in `xs` but ignores all heads of `xs` and all of `zs`. In a context which is head strict (second line), `reverse` is tail strict in `xs`, evaluates tail before head in both `xs` and `zs`, and is head strict in both.

In a head and tail strict context (two last lines), `reverse` is strict in both `xs` and `zs`, and `xs` is in a head and tail strict context which evaluates tail before head, no matter whether the context of `reverse` evaluates head or tail first. Intuitively, `reverse` needs to traverse the whole spine of `xs` before the first constructor in the reversed list is returned. However, `zs` is evaluated in the order as the result of the application.

The argument eod $\{\langle 1,2,\varepsilon\rangle,\langle 1,\varepsilon,2\rangle\}$ says that `reverse` is strict in both `xs` and `zs`, but that we cannot be sure `zs` is evaluated before `reverse` returns a whnf.

The function append

```
append :: nlist × nlist → nlist
append xs zs = case xs of
                 Nil       => zs
                 Cons y ys => Cons y (append ys zs)
```

Analysis results:

$$\begin{array}{lll}
\texttt{append} & : & \{\langle 1,\varepsilon,2\rangle,\langle 1,2,\varepsilon\rangle\}, \\
 & & \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}} \times \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}} \\
\texttt{append} & : & \{\langle 1,\varepsilon,2\rangle,\langle 1,2,\varepsilon\rangle\}, \\
 & & \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}} \times \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}} \\
\texttt{append} & : & \{\langle 1,\varepsilon\rangle,\langle 1,\varepsilon,2\rangle,\langle 1,2,\varepsilon\rangle\}, \\
 & & \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1\rangle,\langle\varepsilon,1,2\rangle\}} \times \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1\rangle,\langle\varepsilon,1,2\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1\rangle,\langle\varepsilon,1,2\rangle\}} \\
\texttt{append} & : & \{\langle 1,\varepsilon\rangle,\langle 1,\varepsilon,2\rangle,\langle 1,2,\varepsilon\rangle\}, \\
 & & \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2\rangle,\langle\varepsilon,1,2\rangle\}} \times \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2\rangle,\langle\varepsilon,1,2\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2\rangle,\langle\varepsilon,1,2\rangle\}}
\end{array}$$

The four lines show that the context of `(append xs zs)` is transmitted to the arguments. The first line does not say that all elements of the first sublist have been evaluated when the evaluation of the first element of the second sublist begins. This illustrates that some information about evaluation order between substructures of different data structures is thrown away in our concept of evaluation order type.

The function concat

```
nlist     ::= Nil | Cons nat nlist
nlistlist ::= ListNil | ListCons nlist nlistlist
concat :: nlistlist -> nlist
concat xss = case xss of
               ListNil        => Nil
               ListCons ys yss => append ys (concat yss)
```

Analysis results:

$$\begin{array}{lll}
\texttt{concat} & : & \{\langle 1,\varepsilon\rangle\}, \texttt{nlistlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\},\texttt{ListCons}:\{\langle\varepsilon,1,2\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,1,2\rangle\}} \\
\texttt{concat} & : & \{\langle 1,\varepsilon\rangle\}, \texttt{nlistlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\},\texttt{ListCons}:\{\langle\varepsilon,1,2\rangle\}} \rightarrow \texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon,2,1\rangle\}}
\end{array}$$

Note that the list of lists is evaluated head before tail, regardless of the order in which the result is traversed.

6.5 Applications: Optimization of suspensions

Bloss describes how evaluation order analysis (as found by Bloss and Hudak's path analysis) can be used for optimizing suspensions in a lazy language [9] [6, Sec. 4.3]. Here we summarize Bloss's description.

When $\mathtt{x}$ is a variable, we let the symbols $\mathtt{x}^j$ and $\mathtt{x}^i$ range over occurrences of $\mathtt{x}$ in $\mathtt{e}$. Once the function description ζ is found as described in Section 6.3.5, it is easy to find the order in which the different occurrences of a variable are accessed. Given expression $\mathtt{e}$, define $\mathtt{e}'$ to be an expression identical to $\mathtt{e}$ except that each occurrence of a free variable $\mathtt{x}$ is replaced by a unique free variable $\mathtt{x}^i$. Now compute

$$\Pi_{occ} = \mathcal{R}[\![\, \mathtt{e}' \,]\!]\zeta\tau$$

to find the variable occurrence paths through $\mathtt{e}$ in context τ.

The following table describes the six possibilities for the status of a variable $\mathtt{x}$ at an occurrence $\mathtt{x}^i$. The six possibilities are spanned by the three possibilities for the variable's past (definitely not already evaluated, definitely already evaluated, don't know) and two possibilities for its future (definitely no later use, possibly a later use). Five of the six cases allow to improve on the default code for a variable access, and evaluation order analysis may allow us to identify such cases. The default code for a variable access is:

```
if   <status = already evaluated>
then return value
else evaluate;
     overwrite;
     return value
```

Let a set Π_{occ} of paths of variable occurrences through an expression $\mathtt{e}$ be given, and consider the variable occurrence $\mathtt{x}^i$ in $\mathtt{e}$. Any path $\pi \in \Pi_{occ}$ in which $\mathtt{x}^i$ occurs can be split into the part $\pi_{\mathtt{x}^i 1}$ before $\mathtt{x}^i$ and the part $\pi_{\mathtt{x}^i 2}$ after $\mathtt{x}^i$. Let $\Pi_{\mathtt{x}^i 1}$ denote the set of such "before" paths, and let $\Pi_{\mathtt{x}^i 2}$ denote the set of "after" paths.

If no path in $\Pi_{\mathtt{x}^i 1}$ contains any $\mathtt{x}^j$, then $\mathtt{x}$ is definitely not already evaluated at $\mathtt{x}^i$. Conversely, if every path in $\Pi_{\mathtt{x}^i 1}$ contains some $\mathtt{x}^j$, then $\mathtt{x}$ is definitely already evaluated at $\mathtt{x}^i$. Similarly, if no path in $\Pi_{\mathtt{x}^i 2}$ contains any $\mathtt{x}^j$, then there is definitely no use of $\mathtt{x}$ after $\mathtt{x}^i$. The following table shows how to exploit this information.

Optimization of code at occurrence $\mathtt{x}^i$	$\mathtt{x}$ already evaluated	$\mathtt{x}$ not yet evaluated	$\mathtt{x}$ possibly already evaluated
No later use of $\mathtt{x}$	No status check, No evaluation, No overwrite	No status check, No overwrite	No overwrite
Possibly a later use of $\mathtt{x}$	No status check, No evaluation, No overwrite	No status check	No optimization possible

The method presented in this paper allows to obtain a useful analysis where lazy data structures are involved. Previous analyses, such as Bloss and Hudak's, do not deal with data structures and thus are not sufficiently powerful to obtain useful results for such simple cases as the `from` function:

```
from n = Cons n (from (n+1))
```

Assume we evaluate the expression `e` $\equiv$ `(sum (take m (from k)))`, where we use the first version of `take` from the previous section. Then `(from k)` is in the context $\texttt{nlist}_{\texttt{Cons}:\{\langle\varepsilon\rangle,\langle\varepsilon,1,2\rangle\}}$, and we find that `n` is never previously evaluated at the first occurrence, and always previously evaluated at the second. Hence at the first occurrence it is always safe to evaluate the suspension for `n` immediately and overwrite it with the result. At the second occurrence, it is never necessary to evaluate `n` or to overwrite `n` with its value. Notice that `from` is not strict in `n`, so `n` could not be evaluated *prior* to the entry to `from`.

This analysis would not be possible using Bloss and Hudak's methods since the order of evaluation of the body of `from` depends on the context in which the call to `from` appears.

6.6 Occurrence path analysis

Above we have described in detail a variable path analysis which finds a set of variable paths, possibly containing ε, for a given expression and given context. Several similar analyses can be built within the same framework: Evaluation order analyses based on *occurrence paths* or *evaluation order relations*, and a backwards strictness analysis based on *strictness sets*.

To obtain a new analysis within this framework, we must redefine the following items, used in the analysis functions $\mathcal{R}$ and $\mathcal{T}$ already defined.

- A finite lattice $Eod(\texttt{e})$ of evaluation order descriptions for an expression `e`. This was $\text{PathSet}(Vars(\texttt{e}) \cup \{\varepsilon\})$ above.
- A finite lattice $AEod(n)$ for describing the evaluation order of n points. This is the basis for defining application contexts and evaluation order types. This was $\text{PathSet}\{\varepsilon, 1, \ldots, n\}$ above.
- The combination operation $a(r_1, \ldots, r_n)$ for these lattices. This was variable path combination above.
- The plumbing functions R_{var}, R_{An}, and $R_{\texttt{case},n}$ used in analysis function $\mathcal{R}$ to describe the evaluation order of variables, base function applications, and `case`.
- Functions $\mathcal{A}$ and $\mathcal{A}_\varepsilon : Eod(\texttt{e}) \times \text{Var}^n \to AEod(n)$ to extract argument eod's for given variables in expression `e` from an eod on `e`.

The constant part or the "frame" of the framework consists of the recursive definition of ζ, the analysis functions $\mathcal{R}$ and $\mathcal{T}$, and the way application contexts $\alpha \in Actx(t_1 \ldots t_n)$ and uniform evaluation order types $\tau \in Eot(t)$ are built from the elements of $AEod$.

In this section we define an occurrence path analysis which is a refinement on the variable path analysis intended to serve as a bridge to the analysis based on evaluation order relations. That analysis in turn is interesting because it is cheaper than the variable path analysis.

6.6.1 Occurrence paths and their operations

An *occurrence* in an expression $\mathtt{e}$ is a finite sequence of indexes, which are natural numbers. An occurrence p labels a subexpression of $\mathtt{e}$ which we denote by $\mathtt{e}[p]$.

The empty occurrence $p = \varepsilon = []$ labels the entire expression $\mathtt{e}$, the occurrence $p = 1.\varepsilon = [1]$ labels the first subexpression, *etc.* More formally, we define $\mathtt{e}[\varepsilon] \equiv \mathtt{e}$, and for $\mathtt{e} \equiv (\mathtt{op}\ \mathtt{e}_1 \ldots \mathtt{e}_i \ldots \mathtt{e}_n)$, we define $\mathtt{e}[i.p] \equiv \mathtt{e}_i[p]$. Here $\equiv$ denotes syntactical identity.

In examples we are less formal and use integers for labels instead of sequences of integers. We will sometimes use an integer i when we mean a sequence $[i]$ of length one.

Letting $\mathrm{Occ}(\mathtt{e})$ denote the set of occurrences in $\mathtt{e}$, we have $\mathrm{Occ}(\mathtt{op}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n) = \{\varepsilon\} \cup \{\ i.p_i \mid p_i \in \mathrm{Occ}(\mathtt{e}_i)\ \}$. Given a set D of occurrences, an *occurrence path* π is an element of $\mathrm{Path}(D)$, and an *occurrence path set* Π is an element of $\mathrm{PathSet}(D)$.

Interpreting occurrences as events, $\varepsilon \in \mathrm{Occ}(\mathtt{e})$ means precisely the same as for variable paths, namely that the entire expression reaches whnf. The event $p \in \mathrm{Occ}(\mathtt{e})$ means that the subexpression $\mathtt{e}[p]$ reaches whnf, which relates to variable path analysis when $\mathtt{e}[p]$ is a variable occurrence.

6.6.2 Combining occurrence paths

Thus the concepts are much as for variable path sets. Combination of occurrence paths π_i for subexpressions $\mathtt{e}_i$ into occurrence paths for expression $\mathtt{e} \equiv (\mathtt{op}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n)$ is also basically the same operation as before. However, all occurrences p in constituent path π_i must be prefixed by the index i to become occurrences relative to the containing expression $\mathtt{e}$.

Definition 2 Let $\Pi_a \in \mathrm{PathSet}\{\varepsilon, 1, \ldots, n\}$ be a path set, called the *combiner*, and let $\Pi_i \in \mathrm{PathSet}(D_i)$ be occurrence path sets. Then $\Pi = \Pi_a(\Pi_1,\ldots,\Pi_n)$ is an occurrence path set, called the Π_a *combination* of $\Pi_1 \ldots \Pi_n$. It is defined as follows

$$\Pi \ = \ \bigcup \{\ \pi_a(\pi_1,\ldots,\pi_n) \mid \pi_a \in \Pi_a,\ \pi_i \in \Pi_i\ \}$$

where basic occurrence path combination $\pi(\pi_1,\ldots,\pi_n)$ is defined by

$$\begin{array}{lcl} \langle\rangle(\pi_1,\ldots,\pi_n) & = & \{\ \langle\rangle\ \} \\ (\varepsilon :: \pi)(\pi_1,\ldots,\pi_n) & = & \{\ \varepsilon :: \pi' \mid \pi' \in \pi(\pi_1,\ldots,\pi_n)\ \} \\ (i :: \pi)(\pi_1,\ldots,\pi_n) & = & \{\ (i.\pi_{i1}) + \pi' \mid \pi_i = \pi_{i1} +\!\!+ \langle\varepsilon\rangle +\!\!+ \pi_{i2} \text{ and} \\ & & \qquad\qquad \pi' \in (i.\pi_{i2}) \| \pi(\pi_1,\ldots,\pi_n)\ \} \end{array}$$

Occurrence prefixing $(i.\pi)$ distributes over all elements of a path, so that $i.\langle p_1, \ldots, p_k\rangle = \langle i.p_1, \ldots, i.p_k\rangle$. Path interleaving $\|$ was defined in Section 6.2.1 above. □

6.6.3 Parametrizing for occurrence paths

The evaluation order description for an expression $\mathtt{e}$ is $Eod(\mathtt{e})$ = PathSet(Occ($\mathtt{e}$)), and the set of argument eod's is $AEod(n) = \text{PathSet}\{\varepsilon, 1, \ldots, n\}$. The evaluation of a variable $\mathtt{x}$ gives a singleton path consisting of the occurrence ε: $R_{var}(\mathtt{x}) = \{\langle\varepsilon\rangle\}$. The evaluation of a base function application can be described by an occurrence path combination R_{An} $= \{\langle 1, \ldots, n, \varepsilon\rangle\}$, and evaluation of a $\mathtt{case}$ expression by $R_{\mathtt{case},n} = \{\langle 0, 1, \varepsilon\rangle, \ldots, \langle 0, n, \varepsilon\rangle\}$.

Extracting an argument eod for $\mathtt{x}_1,\ldots,\mathtt{x}_n$ from an occurrence path set Π for expression $\mathtt{e}$ is done as follows:

$$\mathcal{A},\ \mathcal{A}_\varepsilon : \text{PathSet}(D) \times \text{Var}^n \rightarrow \text{PathSet}\{1, \ldots, n, \varepsilon\}$$
$$\mathcal{A}_\varepsilon(\Pi,(\mathtt{x}_1,\ldots,\mathtt{x}_n)) = \{\ lazyvarindex(\pi) \mid \pi \in \Pi\ \}$$
$$\mathcal{A}(\Pi,(\mathtt{x}_1,\ldots,\mathtt{x}_n)) = \mathcal{A}_\varepsilon(\Pi,(\mathtt{x}_1,\ldots,\mathtt{x}_n)) \setminus \{\varepsilon\}$$

Function *lazyvarindex* computes a path of variable indices i (or ε) describing the order of $\mathtt{x}_1,\ldots,\mathtt{x}_n$ in an occurrence path

$$lazyvarindex\text{: Path}(D) \rightarrow \text{Path}\{\varepsilon, 1, \ldots, n\}$$
$$\begin{array}{lcll} lazyvarindex(\langle\rangle) & = & \langle\rangle & \\ lazyvarindex(\varepsilon :: \pi) & = & \langle\varepsilon\rangle + lazyvarindex(\pi) & \\ lazyvarindex(p :: \pi) & = & \langle i\rangle + lazyvarindex(\pi) & \text{if } \mathtt{e}[p] \equiv \mathtt{x}_i \\ & = & lazyvarindex(\pi) & \text{otherwise} \end{array}$$

It is necessary to refer to the expression $\mathtt{e}$ (from which the path was computed) to test whether an occurrence p is a variable occurrence. Note that we rely on the path concatenation operation "+" to delete all but the first occurrence of an index i from the result.

6.6.4 Relation to variable path analysis

The occurrence path analysis is interesting because it records the order of evaluation of individual variable occurrences, not only the first evaluation of a variable. However, the main role for the occurrence path analysis is to serve as a bridge between the analysis based on variable paths presented above and the analysis based on evaluation order relations presented below in Section 6.7.

We shall not here formalize the connection between these analyses, but will point out that the variable path analysis seems intuitively very plausible and that the concepts have precedents in the literature [8][6]. The variable path analysis hopefully prepared the reader for the less traditional but in some sense more fundamental occurrence path analysis. The occurrence path analysis in turn offers an explanation of the ε used in the variable path analysis. More importantly, it allows to view the evaluation order relations (used below) as assertions about the occurrence path set for an expression, thus providing a model for evaluation order relations.

Let $\mathtt{e}$ be an expression and let π be an occurrence path for $\mathtt{e}$. We must consider path elements p for which $\mathtt{e}[p]$ is a variable $\mathtt{x}$.

$$\begin{array}{lcll} \multicolumn{4}{l}{\mathit{lazyvar}\colon \mathrm{Path}(\mathrm{Occ}(\mathtt{e})) \to \mathrm{Path}(\mathit{Vars}(\mathtt{e}) \cup \{\varepsilon\})} \\ \mathit{lazyvar}(\langle\rangle) & = & \langle\rangle & \\ \mathit{lazyvar}(\varepsilon :: \pi) & = & \langle\varepsilon\rangle + \mathit{lazyvar}(\pi) & \\ \mathit{lazyvar}(p :: \pi) & = & \langle\mathtt{x}\rangle + \mathit{lazyvar}(\pi) & \text{if } \mathtt{e}[p] \equiv \mathtt{x} \\ & = & \mathit{lazyvar}(\pi) & \text{otherwise} \end{array}$$

A small notational problem arises for an expression $\mathtt{e} \equiv \mathtt{x}_i$ consisting only of a variable occurrence. The (only) occurrence path for this is $\langle\varepsilon\rangle$, and $\mathit{lazyvar}(\varepsilon)$ is $\langle\mathtt{x}_i\rangle$, not $\langle\mathtt{x}_i,\varepsilon\rangle$ as desired. In general the variable path corresponding to an occurrence path π is therefore $\mathit{lazyvar}(\pi) + \langle\varepsilon\rangle$. This is annoying as the paths computed by *lazyvar* will contain ε in all other cases (so in particular duplication-free concatenation of ε will have no effect in all other cases).

6.7 Evaluation order relations

Another analysis in the framework (in fact, the first one to inhabit it) is based on evaluation order relations, and is interesting because it is less expensive (and less precise) than the previous analyses.

An *evaluation order relation (eor)* on the set D of occurrences is a pair $r = (r_s, r_w)$ of reflexive relations on D for which $r_s \subseteq r_w$. Thus the set $\mathit{Eor}(D)$ of evaluation order relations on D is

$$a, r \in \mathit{Eor}(D) = \{ (r_s, r_w) \mid I_D \subseteq r_s \subseteq r_w \subseteq D^2 \}$$

where $I_D = \{ (p, p) \mid p \in D \}$ is the identity relation on D. We order the set $\mathit{Eor}(D)$ of eor's on D by componentwise reverse set inclusion, so

$$r_1 \sqsubseteq_E r_2 \text{ if and only if } r_{1s} \supseteq r_{2s} \text{ and } r_{1w} \supseteq r_{2w}$$

With this ordering $\mathit{Eor}(D)$ is a lattice. It is easy to see that its height is at least $n(n-1)$ and at most $2n(n-1)$, in short $O(n^2)$, where $n = |D|$ is the number of elements of D. The number of elements of $\mathit{Eor}(D)$ is not greater than $2^{O(n)}$. Least upper bound $\sqcup_E$ is componentwise relation intersection, and greatest lower bound $\sqcap_E$ is componentwise relation union. The least element of the lattice is the complete relation $\perp_E = (r_s, r_w) = (D^2, D^2)$ in which all occurrences are related by r_s (and hence by r_w); the greatest element is $\top_E = (I_D, I_D)$ which relates every expression only to itself.

For $r = (r_s, r_w)$ we shall write the relation r_s as $\xrightarrow{r}$ and the relation r_w as $\overset{r}{\dashrightarrow}$. The relation r_s (or $\xrightarrow{r}$) is called *strongly precedes* and r_w (or $\overset{r}{\dashrightarrow}$) is called *weakly precedes*. Naturally, the inverse relations r_s^{-1} and r_w^{-1} are written as reversed arrows.

When clear from the context we will leave out the superscript r on arrows. The lattice $\mathit{Eor}\{1, 2\}$ of evaluation order relations on the two-element set $\{1, 2\}$ is shown in Figure 1 using the arrow notation.

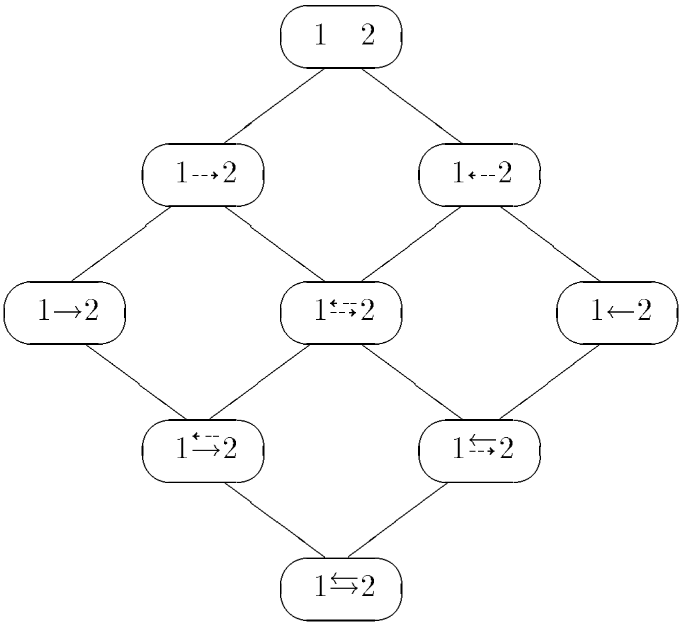

Figure 1: The lattice of two-point evaluation order relations

6.7.1 Evaluation order

Let $\mathtt{e}_1$ and $\mathtt{e}_2$ be two subexpression occurrences. The assertion $\mathtt{e}_1 \rightarrow \mathtt{e}_2$ means: if $\mathtt{e}_2$ is evaluated at all, then $\mathtt{e}_1$ has been evaluated *before* $\mathtt{e}_2$. Here and elsewhere, "before" means "no later than". Correspondingly, "after" means "strictly later than".

The assertion $\mathtt{e}_1 \dashrightarrow \mathtt{e}_2$ means: either $\mathtt{e}_1$ is not evaluated at all, or is it evaluated before $\mathtt{e}_2$. In other words: $\mathtt{e}_1$ is *not* evaluated after $\mathtt{e}_2$.

The following special cases are particularly interesting. If we have $\mathtt{e}_1 \dashrightarrow \mathtt{e}_2$ as well as the converse $\mathtt{e}_2 \dashrightarrow \mathtt{e}_1$, then at most one of the subexpressions can be evaluated. If in addition $\mathtt{e}_1 \rightarrow \mathtt{e}_2$, then at most $\mathtt{e}_1$ can be evaluated. If $\mathtt{e}_1 \rightarrow \mathtt{e}_2$ and $\mathtt{e}_2 \rightarrow \mathtt{e}_1$ and $\mathtt{e}_1$ and $\mathtt{e}_2$ are *distinct*, then none of them can be evaluated.

Example 5 Consider the program `copy` where we have superscripted expressions with labels.

```
copy xs = (case xs⁶ of
             Nil       => Nil⁵
             Cons y ys => (Cons y¹ (copy ys²)³)⁴
          )⁷
```

To express that the root expression (labelled 6) is evaluated before each of the branches, we write $6 \rightarrow 5$ and $6 \rightarrow 4$. To express that it must reach a whnf before the entire `case` expression does, we write $6 \rightarrow 7$. That the branches are mutually exclusive is expressed by

$4 \mathrel{\substack{\dashleftarrow \\ \dashrightarrow}} 5$. Note that without knowing the context for an application `(copy xs)` of `copy` we are unable to relate, say, 1 and 3. □

6.7.2 Combining evaluation order relations

The evaluation order relation r on a composite expression $\mathtt{e} \equiv$ `(op` $\mathtt{e}_1$ ... $\mathtt{e}_n$`)` is expressible as a combination of the evaluation order relations $r_1 \ldots r_n$ on each of its components $\mathtt{e}_1$ ... $\mathtt{e}_n$. The way $r_1 \ldots r_n$ are combined of course depends on which kind of expression `e` is: base function application, constructor application, defined function application, *etc.*, but the general mechanism is the same.

Definition 3 Let $a \in \mathit{Eor}\{\varepsilon, 1, \ldots, n\}$ be an evaluation order relation, called the *combiner*, and let $r_i \in \mathit{Eor}(D_i)$ be evaluation order relations, $i = 1, \ldots, n$. Then $r = a(r_1, \ldots, r_n)$ is an evaluation order relation in $\mathit{Eor}(\{\varepsilon\} \cup \{\ i : p_i \mid i = 1, \ldots, n;\ p_i \in D_i\ \})$, called the eor *combination* of $r_1, \ldots, r_n$. It is defined as follows:

$$
\begin{array}{ll}
\varepsilon \xrightarrow{r} \varepsilon & \\
\varepsilon \xrightarrow{r} i{:}p & \text{if } \varepsilon \xrightarrow{a} i \text{ and } \varepsilon \xrightarrow{r_i} p \\
i{:}p \xrightarrow{r} i{:}q & \text{if } p \xrightarrow{r_i} q \\
i{:}p \overset{r}{\dashrightarrow} i{:}q & \text{if } p \overset{r_i}{\dashrightarrow} q \\
\varepsilon \overset{r}{\dashrightarrow} i{:}p & \text{if } \varepsilon \overset{a}{\dashrightarrow} i \text{ and } \varepsilon \xrightarrow{r_i} p \\
i{:}p \xrightarrow{r} \varepsilon & \text{if } p \xrightarrow{r_i} \varepsilon \text{ and } i \xrightarrow{a} \varepsilon \\
i{:}p \overset{r}{\dashrightarrow} \varepsilon & \text{if } p \overset{r_i}{\dashrightarrow} \varepsilon \text{ and } i \xrightarrow{a} \varepsilon \\
i{:}p \xrightarrow{r} j{:}q & \text{if } p \xrightarrow{r_i} \varepsilon \text{ and } i \xrightarrow{a} j \text{ and } \varepsilon \xrightarrow{r_j} q \\
 & \quad \text{or } i \neq j \text{ and } p \xrightarrow{r_i} \varepsilon \text{ and } i \xrightarrow{a} j \\
i{:}p \overset{r}{\dashrightarrow} j{:}q & \text{if } p \overset{r_i}{\dashrightarrow} \varepsilon \text{ and } i \xrightarrow{a} j \text{ and } \varepsilon \xrightarrow{r_j} q \\
 & \quad \text{or } i \neq j \text{ and } p \overset{r_i}{\dashrightarrow} \varepsilon \text{ and } i \xrightarrow{a} j \\
i{:}p \overset{r}{\dashrightarrow} j{:}q \overset{r}{\dashrightarrow} i{:}p & \text{if } i \neq j \text{ and } i \overset{a}{\dashrightarrow} j \overset{a}{\dashrightarrow} i
\end{array}
$$

Whenever $p \xrightarrow{r} q$, include also $p \overset{r}{\dashrightarrow} q$ to ensure r is an eor. □

6.7.3 Parametrizing for evaluation order relations

We take the evaluation order descriptions for an expression `e` to be eor's on the occurrences in `e`, $\mathit{Eod}(\mathtt{e}) = \mathit{Eor}(\mathrm{Occ}(\mathtt{e}))$, and argument evaluation order relations to be eor's on $D = \{\varepsilon, 1, \ldots, n\}$, so $\mathit{AEod}(n) = \mathit{Eor}\{\varepsilon, 1, \ldots, n\}$. Again, for constructors we shall essentially leave out the ε form argument eod's. Combination was defined above.

The "plumbing" functions for the evaluation order analysis are as follows: The function R_{var} for analysing a variable occurrence is

$$R_{var}(\mathtt{x}) \ = \ \varepsilon \rightarrow \varepsilon$$

The function R_{An} for computing the evaluation order relation on a base function application can conveniently be expressed as a combination of evaluation order relations

$$R_{An} \ = \ \{i \rightarrow \varepsilon \mid i = 1, \ldots, n\} \cup \{i \rightarrow j \mid 1 \leq i < j \leq n\}$$

The function $R_{\mathsf{case},n}$ for computing the evaluation order relation on a `case` expression can also be expressed as a combination of evaluation order relations

$$R_{\mathsf{case},n} \;=\; \{0{\rightarrow}\varepsilon,\, 0{\rightarrow}i,\, i{\dashrightarrow}\varepsilon,\, i\underset{\dashrightarrow}{\dashleftarrow}k \mid i,k \in \{1,\ldots,n\}\ \}$$

We also need to define the functions $\mathcal{A}$ and $\mathcal{A}_\varepsilon$ for extracting argument eod's for given expression `e` and variables $\mathtt{x}_1\ \ldots\ \mathtt{x}_n$.

These functions work by looking at the order of evaluation of variable occurrences in `e`. Thus a notational device is needed for quantifying over occurrences in `e`. For $p \in \{\ \varepsilon,\ 1,\ \ldots,\ n\ \}$ we let p^j range over occurrences (in `e`) of the object associated with p. More precisely, if $p = \varepsilon$, then the entire expression `e` is associated with p, and ε (now *qua* occurrence in `e`) is the only possible value of p^i. If $p = k$, then the variable $\mathtt{x}_k$ is associated with p, and p^i ranges over all occurrences of $\mathtt{x}_k$ in `e`.

Now let expression `e`, variables $\mathtt{x}_1\ \ldots\ \mathtt{x}_n$, and an evaluation order relation r on `e` be given. We want to obtain $a = \mathcal{A}(\mathtt{e},(\mathtt{x}_1\ \ldots\ \mathtt{x}_n),r) \in \mathit{Eor}\{\ \varepsilon,\ 1,\ \ldots,\ n\ \}$. Here ε labels the entire expression `e`, and i labels variable $\mathtt{x}_i$. We describe the analysis by defining $\xrightarrow{a}$ and $\overset{a}{\dashrightarrow}$ in terms of the relation r on occurrences of `e`.

Definition 4 The argument eor $a = \mathcal{A}_\varepsilon(r,(\mathtt{x}_1\ \ldots\ \mathtt{x}_n)) \in \mathit{AEod}(n)$ is defined as follows:
$p \xrightarrow{a} q$ iff $\forall q^i.\exists p^j.p^j \xrightarrow{r} q^i$
$p \overset{a}{\dashrightarrow} q$ iff $\forall q^i.((\forall p^j.p^j \overset{r}{\dashrightarrow} q^i) \vee (\exists p^j.p^j \xrightarrow{r} q^i))$
Intuitively, the object associated with p is evaluated before that associated with q if to every occurrence of q there is an occurrence of p which is evaluated before.

The object associated with p is not evaluated after that associated with q if to every occurrence of q, either all occurrences of p are not evaluated after it, or there is one which is evaluated before. □

Two special cases are worth noting. If variable $\mathtt{x}_k$ does not occur in `e`, then $p \xrightarrow{a} k$ for all $p \in \{\ \varepsilon,\ 1,\ \ldots,\ n\ \}$, and $k \overset{a}{\dashrightarrow} \varepsilon$. Furthermore, if variable $\mathtt{x}_k$ does not occur but variable $\mathtt{x}_h$ does, then $k \overset{a}{\dashrightarrow} h$.

Definition 5 The argument eor $a = \mathcal{A}(r,(\mathtt{x}_1\ \ldots\ \mathtt{x}_n)) \in \mathit{AEod}(n)$ is defined to be the restriction $\mathcal{A}_\varepsilon(r,(\mathtt{x}_1\ \ldots\ \mathtt{x}_n)) \setminus \{\varepsilon\}$ to non-ε occurrences. □

6.7.4 Relation to occurrence paths

An evaluation order relation such as $1{\rightarrow}2$ can be taken to denote a set of occurrence paths. Conversely, a set of occurrence paths can be approximated by an evaluation order relation.

There is a concretization function γ_E from evaluation order relations to sets of paths, definable as follows:

$$\gamma_E\colon\ \mathit{Eor}(D) \rightarrow \mathrm{PathSet}(D)$$

$$\begin{array}{lcl}
\gamma_E(j{\rightarrow}i) & = & \{\langle\ldots\rangle\} \cup \{\langle\ldots j\ldots\rangle\} \cup \{\langle\ldots j\ldots i\ldots\rangle\} \\
\gamma_E(j{\dashrightarrow}i) & = & \{\langle\ldots\rangle\} \cup \{\langle\ldots j\ldots\rangle\} \cup \{\langle\ldots i\ldots\rangle\} \cup \{\langle\ldots j\ldots i\ldots\rangle\} \\
 & & \text{where } i \text{ and } j \text{ are distinct} \\
 & & \text{and } \ldots \text{ denotes path fragments in } (D \setminus \{i,j\})^*. \\
\gamma_E(r_1 \cup r_2) & = & \gamma_E(r_1) \cap \gamma_E(r_2)
\end{array}$$

Since greatest lower bound $\sqcap_E$ in $Eor(D)$ is $\cup$, and greatest lower bound $\sqcap_P$ in PathSet(D) is $\cap$, γ_E is a $\sqcap$-morphism by definition, and it is co-strict: it maps $\top_E$ to $\top_P$. Hence it is monotonic and has an approximate inverse, which is the abstraction function α_E mapping path sets to the best approximating evaluation order relations. The abstraction function α_E is definable as follows: (This result seems to follow from [14], cf. [65]).

$$\begin{array}{lcl} \alpha_E & : & \mathrm{PathSet}(D) \to Eor(D) \\ \alpha_E(\Pi) & = & \bigsqcap_E \{ \ r \in Eor(D) \mid \Pi \sqsubseteq_P \gamma_E(r) \ \} \end{array}$$

With this definition, α_E is monotonic and it is easy to check that:

$$\gamma_E(\alpha_E(\Pi)) \sqsupseteq_P \Pi \ \text{ for all } \Pi \in \mathrm{PathSet}(D)$$

and

$$\alpha_E(\gamma_E(r)) \sqsubseteq_E r \ \text{ for all } r \in Eor(D)$$

Abstracting a path set and concretizing it again gives a bigger path set (less information). When concretizing an evaluation order relation r and abstracting it again we would expect to obtain r again. But γ_E is not injective, so this is not the case. However, it is easy to see that $\gamma_E(\alpha_E(\gamma_E(r))) = \gamma_E(r)$, so the result is safe. (In other words, α_E and γ_E form a Galois connection rather than a Galois insertion [45] [65]).

To see that γ_E is not injective, consider $D = \{1,2,3\}$ and the evaluation order relations $1 \leftrightarrows 2 \rightarrow 3$ and $1 \leftrightarrows 2 \leftrightarrows 3$. Both are mapped to the singleton path set $\{\langle\rangle\}$ by γ_E. This non-injectiveness of γ_E seems to indicate that the lattice of evaluation order relations has some irrelevant elements; it is not a perfect abstraction of PathSet(D).

However, one can see evaluation order relations as a more efficient but less precise way of computing some of the information that can be computed by backwards path analysis. Notice that this is done while retaining much the same framework: same notion of evaluation order types, same analysis functions, same way of combining evaluation order information for subexpressions into evaluation order information for the containing expression, *etc.*

To exemplify the loss of expressiveness, we remark that the concretization function γ_E maps evaluation order relations to prefix-closed sets of paths. This means that information like "if `x` is evaluated now, then `y` will definitely be evaluated later" cannot be expressed with evaluation order relations. It *can* be expressed with path sets.

6.7.5 Complexity of evaluation order relations

The complexity of the analysis based on evaluation order relations is much lower than that of the variable path analysis. The height of the lattice $AEod(n)$ is $O(n^2)$ where it is worse than $n!$ for variable path analysis. Specifically, $AEod(0) = Eor\{\varepsilon\} = \{\varepsilon \rightarrow \varepsilon\}$ has height 0 and 1 element; $AEod(1)$ has height 4 and 9 elements; and $AEod(2)$ has height 10.

Thus $Eot(\texttt{nat}) = AEod(0)$ has height 0 and 1 element, and lattice $Eot(\texttt{nlist}) = AEod(0) \times AEod(2) \times AEod(0)$ is isomorphic to $AEod(2) = Eor\{\varepsilon,1,2\}$. Since we ignore the ε element for constructor argument descriptions, $Eot(\texttt{nlist})$ is actually isomorphic to the lattice shown in Figure 1.

It follows that $Eot(\texttt{nlist})$ has height 4 and has 9 elements, so the lattice of function descriptions for `take` has height only $9 * (11 + 4 + 0) = 135$ when using evaluation order relations as compared to 2432 for variable path sets. More importantly, the height of the abstract function domain grows "only" as 2^{m^2} where m is the largest constructor arity in the result type, as compared to $2^{m!}$ for variable path analysis.

6.8 Backwards strictness analysis

Another analysis which can be defined in the same framework is a backwards strictness analysis for data structures. Instead of variable path sets or evaluation order relations, one works with sets of variables. A *strictness set* $s \subseteq D \setminus \{\varepsilon\}$ is a set of variables needed by an expression in a given context. A variable is *needed* if (at least) its whnf must be found for the expression to produce as much of its value as required by the context. Strictness sets do not contain ε since we always assume the whnf of the expression under consideration is needed. When s is the result of analysing an expression in a given context, $\texttt{x} \in s$ means that variable `x` is needed by that expression in that context.

Given a set D of variables, the domain of strictness sets is

$$s \in \mathrm{Strict}(D) = \wp(D \setminus \{\varepsilon\})$$

With the superset ordering $\sqsubseteq_S = \supseteq$, $\mathrm{Strict}(D)$ is a lattice of height n with 2^n elements, where $n = |D|$ is the number of elements of D. The least (most informative) element is $\bot_S = D$ and the greatest (least informative) element is $\top_S = \{\}$. Least upper bound is $\sqcup_S = \cap$ and greatest lower bound is $\sqcap_S = \cup$.

6.8.1 Combining strictness sets

Combination of strictness sets is analogous to combination of variable path sets and is defined as follows.

Definition 6 Let $s_a \in \mathrm{Strict}\{1, \ldots, n\}$ be a strictness set, called the *combiner*, and let $s_i \in \mathrm{Strict}(D_i)$ be strictness sets, $i = 1, \ldots, n$. Then $s = s_a(s_1, \ldots, s_n)$ is a strictness set in $\mathrm{Strict}(\bigcup_{i=1}^n D_i)$, called the *combination* of $s_1, \ldots, s_n$. It is defined as follows:

$$s = \bigcup \{ s_i \mid i \in s_a \}$$

The combination is the union of those strictness sets s_i which are needed according to the combiner s_a. □

6.8.2 Parametrizing for backwards strictness analysis

We put $Eod(\mathtt{e}) = \mathrm{Strict}(Vars(\mathtt{e}))$ where $Vars(\mathtt{e})$ is the set of variables occurring in e, and $AEod(n) = \mathrm{Strict}\{1, \ldots, n\}$. The set of variables needed to find the whnf of variable x is $\{\mathtt{x}\}$, so R_{var} must be

$$R_{var}(\mathtt{x}) \;=\; \{\mathtt{x}\}$$

The set of needed variables in a base function application is the union of the sets of variables needed in each of the subexpressions, so R_{An} must be

$$R_{An}(s_1, \ldots, s_n) \;=\; s_1 \cup \ldots \cup s_n$$

This corresponds precisely to letting R_{An} be the combiner $\{1, \ldots, n\}$. The set of variables needed to find the whnf of a `case` expression is the set needed to evaluate the root expression, together with those variables which are needed in *every* branch of the `case`, so $R_{\mathsf{case},n}$ must be

$$R_{\mathsf{case},n}(s_0, s_1, \ldots, s_n) \;=\; s_0 \cup (s_1 \cap \ldots \cap s_n)$$

This cannot be expressed very well as a combination of strictness sets, because the strictness sets cannot express disjunctive properties such as "*either* this branch *or* that branch is evaluated". The best approximation would be $R_{\mathsf{case},n} = \{0\}$, which would lose all information about variables needed in every branch.

The argument eod extraction functions simply find the set of indices of strict variables. Moreover, we do not distinguish argument eod's for constructors from those for functions, so $\mathcal{A}_\varepsilon = \mathcal{A}$ is defined as follows:

$$\mathcal{A}(s,(\mathtt{x}_1,\ldots,\mathtt{x}_n)) \;=\; \{\ i \mid \mathtt{x}_i \in s\ \}$$

6.8.3 Relation to variable path analysis

Let $vars(\pi)$ be the set of non-epsilon elements in path π. The connection between variable path sets and strictness sets is the following. A variable x is needed in a variable path π if it occurs in the path, that is $\mathtt{x} \in vars(\pi)$. (Note that it does not matter whether the variable appears before or after ε in the path). Variable x is needed in a path set Π if it occurs in all paths π in the set, that is, $\mathtt{x} \in \bigcap \{\ vars(\pi) \mid \pi \in \Pi\ \}$.

We can define a concretization function γ_S which maps a strictness set s to the set of those paths containing all the variables in s.

$$\gamma_S \quad : \ \mathrm{Strict}(D) \to \mathrm{PathSet}(D)$$
$$\gamma_S(s) \;=\; \{\ \pi \in \mathrm{Path}(D) \mid s \subseteq vars(\pi)\ \}$$

Now $\gamma_S(s_1 \sqcap_S s_2) = \gamma_S(s_1 \cup s_2) = \gamma_S(s_1) \cap \gamma_S(s_2) = \gamma_S(s_1) \sqcap_E \gamma_S(s_2)$, so γ_S is $\sqcap$-morphism. Also, $\gamma_S(\top_S) = \gamma_S(\{\}) = \mathrm{PathSet}(D) = \top_P$, so γ_S has a corresponding best abstraction function α_S, definable as follows.

$$
\begin{array}{lcl}
\alpha_S & : & \mathrm{PathSet}(D) \to \mathrm{Strict}(D) \\
\alpha_S(\Pi) & = & \bigsqcap_S \{ \ s \in \mathrm{Strict}(D) \mid \Pi \sqsubseteq_S \gamma_S(s) \ \} \\
 & = & \bigcup \{ \ s \in \mathrm{Strict}(D) \mid \{\pi {\in} \mathrm{Path}(D) \mid s \subseteq vars(\pi)\} \subseteq \Pi\} \\
 & = & \bigcup \{ \ s \in \mathrm{Strict}(D) \mid \forall \pi \in \Pi . s \subseteq \ vars(\pi) \ \} \\
 & = & \{ \ p \in (D \setminus \{\varepsilon\}) \mid \forall \pi \in \Pi . p \in \ vars(\pi) \ \} \\
 & = & \bigcap \{ \ vars(\pi) \mid \pi {\in} \Pi \ \}
\end{array}
$$

The strictness set $s = \alpha_S(\Pi)$ corresponding to a path set Π is the set of variables that occur in every path in Π.

6.8.4 Examples of backwards strictness analysis

Consider the strictness evaluation order types over `nlist`. As argued above, these correspond to the strictness sets in Strict{1,2}, namely, the descriptions of `Cons`.

A *head strict* context is described by {1}; this is also the context induced by the printer. A *tail strict* context is described by {2}, and a head and tail strict context by {1,2}. Strictness sets as defined here cannot describe disjunctive information, such as a context which will evaluate either head or tail, but not necessarily both. Strictness powersets as defined in Section 6.8.6 below do handle disjunctive information, however.

Below we refer to the example programs listed in Section 6.4 above.

The function length

$$\texttt{length} : \{1\}, \texttt{nlist}_{\texttt{Cons}:\{2\}} \to \texttt{nat}$$

The argument strictness set {1} shows that `length` is strict, and the context $\texttt{nlist}_{\texttt{Cons}:\{2\}}$ for `xs` means that the function is tail strict.

The function sum

$$\texttt{sum} : \{1\}, \texttt{nlist}_{\texttt{Cons}:\{1,2\}} \to \texttt{nat}$$

The `sum` function is strict in `xs` as well as head and tail strict.

The function and

$$\texttt{and} : \{1\}, \texttt{boollist}_{\texttt{Cons}:\{1\}} \to \texttt{bool}$$

The `and` function is strict, and is a natural example of a function which is head strict but not tail strict.

One version of the function take

$$\texttt{take} : \{1\}, \texttt{nlist}_{\texttt{Cons}:\{\}} \times \texttt{nat} \to \texttt{nlist}_{\texttt{Cons}:\{1,2\}}$$

In a head and tail strict context, `take` is strict in `xs` but not `n`, and the context of `xs` is neither head nor tail strict.

Another version of the function take

$$\texttt{take} : \{2\}, \texttt{nlist}_{\text{Cons}:\{1\}} \times \texttt{nat} \rightarrow \texttt{nlist}_{\text{Cons}:\{1,2\}}$$
$$\texttt{take} : \{2\}, \texttt{nlist}_{\text{Cons}:\{1\}} \times \texttt{nat} \rightarrow \texttt{nlist}_{\text{Cons}:\{1\}}$$

This version of `take` is strict in `n` but not `xs`. Like the previous version it is not tail strict on `xs` even in a tail strict context, but it is head strict in a head strict context.

Reverse function with accumulating parameter

$$\texttt{reverse} : \{1\}, \quad \texttt{nlist}_{\text{Cons}:\{2\}} \times \texttt{nlist}_{\text{Cons}:\{\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{\}}$$
$$\texttt{reverse} : \{1\}, \quad \texttt{nlist}_{\text{Cons}:\{2\}} \times \texttt{nlist}_{\text{Cons}:\{1\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{1\}}$$
$$\texttt{reverse} : \{1,2\}, \texttt{nlist}_{\text{Cons}:\{2\}} \times \texttt{nlist}_{\text{Cons}:\{2\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{2\}}$$
$$\texttt{reverse} : \{1,2\}, \texttt{nlist}_{\text{Cons}:\{1,2\}} \times \texttt{nlist}_{\text{Cons}:\{1,2\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{1,2\}}$$

In a simply strict context (first line), `reverse` is strict in `xs`, and `xs` is in a tail strict context. This holds in all the following contexts, too. In a head strict context (second line) `zs` is also in a head strict context. In a tail strict context (third line), `zs` is in a strict and tail strict context. In a head and tail strict context, both `xs` and `zs` are in a strict as well as head and tail strict context.

The function append

$$\texttt{append} : \{1\}, \texttt{nlist}_{\text{Cons}:\{1\}} \times \texttt{nlist}_{\text{Cons}:\{1\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{1\}}$$
$$\texttt{append} : \{1,2\}, \texttt{nlist}_{\text{Cons}:\{2\}} \times \texttt{nlist}_{\text{Cons}:\{2\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{2\}}$$
$$\texttt{append} : \{1,2\}, \texttt{nlist}_{\text{Cons}:\{1,2\}} \times \texttt{nlist}_{\text{Cons}:\{1,2\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{1,2\}}$$

In a head strict context $\{1\}$, `append` is strict in its first argument but not the second, and is head strict in both arguments. In a tail strict context $\{2\}$, `append` is strict in both arguments and tail strict in both arguments. In a head and tail strict context, both arguments to `append` are in a strict as well as head and tail strict context.

The function concat

$$\texttt{concat} : \{1\}, \texttt{nlistlist}_{\text{Cons}:\{1\},\text{ListCons}:\{1\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{1\}}$$
$$\texttt{nlistlist}_{\text{Cons}:\{2\},\text{ListCons}:\{1,2\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{2\}}$$
$$\texttt{nlistlist}_{\text{Cons}:\{1,2\},\text{ListCons}:\{1,2\}} \rightarrow \texttt{nlist}_{\text{Cons}:\{1,2\}}$$

In any strict context, `concat` is strict. In a head strict context, it is also head strict in both levels of lists: `Cons` and `ListCons`. In a tail strict context it is head and tail strict in the list of lists (`ListCons`), but only tail strict in the sublists (`Cons`). In a head and tail strict context, it is head and tail strict in both levels of lists.

6.8.5 Complexity of backwards strictness analysis

The lattice $Eot(\mathrm{nat}) = AEod(0) = \{\{\}\}$ has height 0 and 1 element. The lattice $Eot(\texttt{nlist})$ therefore is isomorphic to $AEod(2) = \mathrm{Strict}\{1{,}2\}$ which has height 2 and 4 elements.

Hence for a function with n arguments, of which m are of type `nlist` and the rest of base type, the application context domain (*i.e.*, the range of its abstract function) has height $1 * (n + 2m) = n + 2m$.

Thus for a function such as `take`, the height of the lattice of function descriptions is $4 * (2 + 2) = 16$, which is reasonable. For `concat`, the height is $4 * 4 = 16$ since the evaluation order type lattice over the result domain `nlist` has 4 elements, and the lattice over the argument domain `nlistlist` has height 4. On the whole we consider this kind of strictness analysis very tractable, although the size of the context domains grows rapidly as the result types get more complex. Thus the height of the abstract function is at least 2^m when m is the largest constructor arity in the result type.

The operations necessary to represent and manipulate the abstract values (strictness sets and the uniform evaluation order types constructed over them) are straightforward to implement and can be quite fast.

6.8.6 A more precise backwards strictness analysis

A more precise backwards strictness analysis for lazy data structures can be obtained by using *strictness powersets*, that is, sets of strictness sets. This way a kind of relational backwards strictness analysis is obtained. One would use the lattice

$$S \;\in\; \mathrm{RStrict}(D) \;=\; \wp(\wp(D \setminus \{\varepsilon\}))$$

with inclusion ordering $\sqsubseteq_R = \subseteq$, least upper bound $\cup$, greatest lower bound $\cap$, least (most informative) element $\bot_R = \{\}$, greatest (least informative) element $\top_R = \wp(D)$, height 2^n and size 2^{2^n} where $n = |D|$ is the number of elements of D.

Combination is defined much as for strictness sets. When $S_a \in \mathrm{RStrict}\{1, \ldots, n\}$ and $S_i \in \mathrm{RStrict}(D_i)$, we obtain $S = S_a(S_1, \ldots, S_n)$ in $\mathrm{RStrict}(\bigcup_{i=1}^{n} D_i)$ as follows

$$S \;=\; \{\; \textstyle\bigcup_{i \in s_a} s_i \mid s_a \in S_a, s_j \in S_j \;\}$$

To obtain an analysis in the general framework, we put $Eod(\texttt{e}) = \mathrm{RStrict}(Vars(\texttt{e}))$ and $AEod(n) = \mathrm{RStrict}\{1, \ldots, n\}$.

The combiner R_{var} is $R_{var}(\texttt{x}) = \{\{\texttt{x}\}\}$. The combiner for n-ary base functions is $R_{An} = \{\{1, \ldots, n\}\}$. In contrast to the simpler strictness analysis presented above, it is now possible to express the strictness of a `case` expression using a combiner, namely $R_{\texttt{case},n} = \{\{0, 1\}, \ldots, \{0, n\}\}$.

This analysis can also be related to the variable path analysis. The concretization function γ_R maps a strictness powerset S to the set of those paths whose variable set belongs to S:

$$\begin{aligned} \gamma_R \quad &: \; \mathrm{RStrict}(D) \to \mathrm{PathSet}(D) \\ \gamma_R(S) \; &= \; \{\; \pi \in \mathrm{Path}(D) \mid vars(\pi) \in S \;\} \end{aligned}$$

It is not hard to see that this is a $\sqcap$-morphism, and that γ_R maps $\top_R$ to $\top_P$. Therefore the abstraction function α_R can be defined as follows:

$$
\begin{array}{lcl}
\alpha_R & : & \mathrm{PathSet}(D) \rightarrow \mathrm{RStrict}(D) \\
\alpha_R(\Pi) & = & \bigsqcap_R \{\ S \in \mathrm{RStrict}(D) \mid \Pi \sqsubseteq_R \gamma_R(S)\ \} \\
 & = & \bigcap \{\ S \in \mathrm{RStrict}(D) \mid \Pi \sqsubseteq_R \{\pi \in \mathrm{Path}(D) \mid \mathit{vars}(\pi) \in S\}\ \} \\
 & = & \bigcap \{\ S \in \mathrm{RStrict}(D) \mid \forall \pi \in \Pi. \mathit{vars}(\pi) \in S\ \} \\
 & = & \bigcap \{\ S \in \mathrm{RStrict}(D) \mid \bigcap \{\mathit{vars}(\pi) \mid \pi \in \Pi\} \in S\ \} \\
 & = & \{\ \mathit{vars}(\pi) \mid \pi \in \Pi\}
\end{array}
$$

The abstraction of a path set Π is the set of sets of variables occurring in some path. Since γ_R is clearly injective, we have $\alpha_R(\gamma_R(S)) = S$ and $\gamma_R(\alpha_R(\Pi)) \sqsupseteq_P \Pi$.

Though less costly than the variable path analysis, this analysis will be very expensive, and we shall not consider it in more detail.

6.9 Recursive data types revisited

In Section 6.2.4.3 we discussed uniform evaluation order types over recursive data types only for the simple example `nlist ::= Nil | Cons nat nlist`. In this section we will study evaluation order types for general recursive data types, and give general versions of the analysis functions $\mathcal{T}$ and $\mathcal{R}$. The procedure we devise here works for all the analyses presented so far. Briefly stated, the task is to find for any given data type t a lattice $\mathit{Eot}(t)$ of evaluation order types which has finite height.

6.9.1 Uniform evaluation order types

There are several reasonable ways to do this, all of which will identify recursive occurrences of a type (or constructor) with structurally "previous" occurrences of the same type or constructor. The most general one seems to be to adapt the definition of $\mathit{Eot}(t)$ already given. We add an argument A which is a set of type names "already seen" in the recursive definition, and when meeting a type already seen, we replace the eot at that point with $\oplus$, pronounced "blank". Thus we redefine Actx and Eot as follows:

$$
\begin{array}{lcll}
\mathit{Actx}(\mathrm{A})(t_1 \ldots t_m) & = & \mathit{AEod}(m) \times \prod_{j=1}^{m} \mathit{Eot}(\mathrm{A})(t_j) & \\
\mathit{Eot}(\mathrm{A})(\mathtt{nat}) & = & \mathit{AEod}(0) & \\
\mathit{Eot}(\mathrm{A})(t) & = & \{\ \oplus\ \} & \text{if } t \in \mathrm{A} \\
\mathit{Eot}(\mathrm{A})(t) & = & \prod_{i=1}^{n} \mathit{Actx}(\mathrm{A} \cup \{t\})\ (t_{i1}\ \ldots\ t_{ic(i)}) & \text{if } t \notin \mathrm{A} \\
 & & \text{where } t ::= C_1\ t_{11} \ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1} \ldots t_{nc(n)} &
\end{array}
$$

We call $\tau \in \mathit{Eot}(\{\})(t)$ a *uniform evaluation order type* over data type t, and call $\alpha \in \mathit{Actx}(\{\})(t_1\ \ldots\ t_m)$ a *uniform application context* over over types $t_1\ \ldots\ t_m$.

It is easy to see that with these modified definitions we have $\mathit{Eot}(\{\})(\mathtt{nlist}) = \mathit{AEod}(0) \times \mathit{AEod}(2) \times \mathit{AEod}(0) \times \{\oplus\}$ which is isomorphic to $\mathit{AEod}(0) \times \mathit{AEod}(2) \times \mathit{AEod}(0)$ as found in Section 6.2.4.3.

Let t be a data type $t ::= C_1\ t_{11} \ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1} \ldots t_{nc(n)}$. A uniform evaluation order type $\tau \in Eot(\{\})(t)$ is a tuple $(\alpha_1, \ldots, \alpha_n)$ of application contexts, one for each constructor C_i. Each application context $\alpha_i = (a_i, (\tau_{i1}, \ldots, \tau_{ic(i)}))$ consists of an argument evaluation order description a_i for constructor C_i, and a $\tau_{ij} \in Eot(\{t\})(t_{ij})$ for each argument. Note that τ_{ij} will not in general be a uniform evaluation order type over the type t_{ij} of the argument; it is less informative than that. In fact, if t_{ij} is t, then τ_{ij} will be (the informationless) element $\oplus$.

6.9.2 Unfolding and folding

In Section 6.2.4.3 above we postulated the operations $\tau @ C_i$ and $t[\alpha_1, \ldots, \alpha_n]$ which are used in the analysis equations for constructor application and for the destructor `case`. The present section defines these operations. First we give an informal explanation, then all the technical details needed for a more formal explanation.

For the analysis of a constructor application (C_i $\mathtt{e}_1$ ... $\mathtt{e}_m$) whose evaluation order type is τ_0, we need the *unfolding* $\tau_0 @ C_i$ of τ_0 at C_i. This is the application context $(a, (\tau_1, \ldots, \tau_m))$ for the constructor application, where a is the argument eod describing the order of evaluation of $\mathtt{e}_1 \ldots \mathtt{e}_m$, and τ_j is the evaluation order type for $\mathtt{e}_j$. Thus to analyse the construction of a data type value, we extract a uniform application context from a uniform evaluation order type. This implies unfolding of eots appearing inside the application context.

Conversely, to analyse a `case` expression, which *destructs* a data type value, we must construct a uniform evaluation order type (for the root expression of the `case`) from uniform application contexts α_j (for the `case` branches). This we call the *folding* $t_0[\alpha_1, \ldots, \alpha_n]$ of application contexts $\alpha_1 \ldots \alpha_n$ with respect to type t_0.

At first, one might think that since a uniform eot over type t_0 has form $\tau_0 = (\alpha_1, \ldots, \alpha_n)$, one could simply define the unfolding $\tau_0 @ C_i$ to be α_i. However, this would in general throw away information because $\alpha_i = (a_i, (\tau_{i1}, \ldots, \tau_{im}))$ is not in $Actx(\{\})(t_{i1} \ldots t_{im})$ but in $Actx(\{t_0\})(t_{i1} \ldots t_{im})$, so τ_{ij} is not a uniform evaluation order type over t_{ij}. The subtrees in α_i corresponding to t_0 are all $\oplus$, that is, informationless. Therefore we need to expand α_i with τ_0 at occurrences of t_0 inside α_i.

Similarly, one might think that the folding $t_0[\alpha_1, \ldots, \alpha_n]$ could be defined simply as the tuple $(\alpha_1, \ldots, \alpha_n)$, but the result would not be a uniform evaluation order type over t_0. The constructors belonging to type t_0 may have different argument evaluation order descriptions associated with them at the various occurrences of t_0 inside the α_j. For the result to be a uniform evaluation order type, a safe approximation to these should be found, such as their least upper bound.

Unfortunately, the formal definition of unfolding and folding requires a number of technical and not very exciting definitions. The reader is probably best advised to skip the remainder of the section.

6.9.3 Normalizing evaluation order types

First we define a generalization of uniform evaluation order types and uniform application contexts, called τ-tree and α-tree.

Definition 7 A τ-tree over t is

- a τ-node labelled t, when t is a base type, or
- a τ-node labelled t with n outgoing labelled edges, where the edge labelled C_i ends in an α-tree over $t_{i1} \ldots t_{ic(i)}$, when t is a data type $t ::= C_1\ t_{11} \ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1} \ldots t_{nc(n)}$, or
- a τ-node labelled "$\oplus$" ("blank").

An α-tree over $t_1 \ldots t_m$ is

- an α-node labelled with an argument eod $a \in AEod(m)$, with m outgoing labelled edges, where the edge labelled j ends in a τ-tree over t_j.

Clearly, τ-trees generalize uniform evaluation order types, and α-trees generalize uniform application contexts. In a uniform evaluation order type there is no path with the same τ-node label (*i.e.*, type name) appearing twice. □

We shall define normalization operations NR_τ and NR_α to *reduce* τ-trees and α-tress which are "too large", to obtain uniform eots and application contexts. This is done by cutting off branches of the trees.

Definition 8 Assume τ-tree τ over t contains a uniform eot over t as a subtree at the root. Then the reduction of τ is $NR_\tau(\{\})(t)\ \tau$, where NR_τ is defined below.

$$
\begin{array}{lll}
NR_\tau(\mathrm{A})(t)\ \tau & = \tau & \text{is } t \text{ is a base type} \\
 & = \oplus & \text{if } t \in \mathrm{A} \\
 & = (NR_\alpha(\mathrm{A}\cup\{t\})(t_{i1} \ldots t_{ic(i)})\ \alpha_i)_{i=1,\ldots,n} & \text{otherwise} \\
 & \quad \text{where } t ::= C_1\ t_{11} \ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1} \ldots t_{nc(n)} & \\
 & \quad \text{and } \tau = (\alpha_1, \ldots, \alpha_n) & \\
NR_\alpha(\mathrm{A})(t_1 \ldots t_m)\ \alpha & = (a,(NR_\tau(\mathrm{A})(t_j)\ \tau_j)_{j=1,\ldots,m}) & \\
 & \quad \text{where } \alpha = (a,(\tau_1,\ldots,\tau_m)) &
\end{array}
$$

It is not hard to see that if τ contains a uniform eot as a subtree at the root, then $NR_\tau(\mathrm{A})(t)\ \tau$ is an element of $Eot(\mathrm{A})(t)$. In particular for $\mathrm{A} = \{\}$, $NR_\tau(\{\})(t)\ \tau$ is a uniform eot over t. Similarly for NR_α. □

Corresponding to the reducing normalization operations NR_τ and NR_α we define normalization operations NE_τ and NE_α to *expand* τ-trees and α-tress which are "too small", to obtain uniform eots and application contexts. This is done by growing new branches containing the least argument eod $\perp_{AEod(m)}$ where necessary to obtain a uniform eot.

Definition 9 Assume τ-tree τ over t is a subtree (at the root) of some uniform eot over t. Then the expansion of τ is $NE_\tau(\{\})(t)\ \tau$, where NE_τ is defined below.

$$
\begin{array}{lll}
NE_\tau(\mathrm{A})(t)\ \tau & = \tau & \text{if } t \text{ is a base type} \\
& = \oplus & \text{if } t \in \mathrm{A} \\
& = (NE_\alpha(\mathrm{A}\cup\{t\})(t_{i1}\ldots t_{ic(i)})\ \perp_i)_{i=1,\ldots,n} & \text{if } t \notin \mathrm{A} \text{ and } \tau = \oplus \\
& \quad \text{where } \perp_i = (\perp_{AEod\,(c(i))},(\oplus,\ldots,\oplus)) & \\
& = (NE_\alpha(\mathrm{A}\cup\{t\})(t_{i1}\ldots t_{ic(i)})\ \alpha_i)_{i=1,\ldots,n} & \text{if } t \notin \mathrm{A} \text{ and } \tau \neq \oplus \\
& \quad \text{where } t ::= C_1\ t_{11}\ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1}\ldots t_{nc(n)} & \\
& \quad \text{and } \tau = (\alpha_1,\ \ldots,\ \alpha_n) & \\
NE_\alpha(\mathrm{A})(t_1\ldots t_m)\ \alpha & = (a,(NE_\tau(\mathrm{A})(t_j)\ \tau_j)_{j=1,\ldots,m}) & \\
& \quad \text{where } \alpha = (a,(\tau_1,\ldots,\tau_m)) &
\end{array}
$$

If τ is a subtree (at the root) of some uniform eot, then $NE_\tau(\mathrm{A})(t)\ \tau$ is an element of $Eot(\mathrm{A})(t)$, and in particular, with $\mathrm{A} = \{\}$, $NE_\tau(\{\})(t)\ \tau$ is a uniform eot over t. Similarly for NE_α. □

6.9.4 Subtrees and substitution

We need a function for replacing an "empty" subtree corresponding to t_0 with some τ-tree over t_0.

Definition 10 Let t_0 be a data type and τ_0 a uniform eot over t_0. The *substitution* of τ_0 at t_0 in a τ-tree τ over t is given by

$$
\begin{array}{lll}
S_\tau(t)(t_0)(\tau_0)\ \tau & = \tau & \text{if } t \text{ is a base type} \\
& = \tau_0 & \text{if } t = t_0 \\
& = (S_\alpha(t_{i1}\ldots t_{ic(i)})(t_0)(\tau_0)\ \alpha_i)_{i=1,\ldots,n} & \text{otherwise} \\
& \quad \text{where } \tau = (\alpha_1,\ \ldots,\ \alpha_n) & \\
& \quad \text{and } t ::= C_1\ t_{11}\ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1}\ldots t_{nc(n)} & \\
S_\alpha(t_1\ldots t_m)(t_0)(\tau_0)\ \alpha & = (a,(S_\tau(t_j)(t_0)(\tau_0)\ \tau_j)_{j=1,\ldots,m}) & \\
& \quad \text{where } \alpha = (a,(\tau_1,\ldots,\tau_m)) &
\end{array}
$$

Note that $S_\tau(t)(t_0)(\tau_0)\ \tau$ is a τ-tree over t, and that if $\tau \in Eot\{t\}(t)$ then $S_\tau(t)(t_0)(\tau_0)\ \tau$ has a uniform eot over t as a subtree at its root. □

Also, we need a function to find the set of those subtrees of τ- or α-trees which correspond to (*i.e.*, give evaluation order information about) a given data type t_0.

Definition 11 Let t_0 be a data type. The set of (expanded) subtrees of τ-tree τ corresponding to type t_0 is the set of uniform evaluation order types over t_0 given by

$$
\begin{array}{lll}
ST_\tau(t)(t_0)\ \tau & = \{\} & \text{if } t \text{ is a base type} \\
& = \{\ NE_\tau(\{\})(t_0)\ \tau\ \} & \text{if } t = t_0 \\
& = \bigcup_{i=1}^{n} ST_\alpha(t_{i1}\ldots t_{ic(i)})(t_0)\ \alpha_i & \text{otherwise} \\
& \quad \text{where } \tau = (\alpha_1,\ \ldots,\ \alpha_n) & \\
& \quad \text{and } t ::= C_1\ t_{11}\ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1}\ldots t_{nc(n)} & \\
ST_\alpha(t_1\ldots t_m)(t_0)\ \alpha & = \bigcup_{j=1}^{m} ST_\tau(t_j)(t_0)\ \tau_j & \\
& \quad \text{where } \alpha = (a,(\tau_1,\ldots,\tau_m)) &
\end{array}
$$

Note that the elements of $ST_\tau(t)(t_0)\ \tau$ are uniform eots over t_0. □

6.9.5 Formal definitions of unfolding and folding

Using the machinery introduced above, we finally have the following definitions.

Definition 12 The *unfolding* of a uniform evaluation order type τ_0 over type t_0 at constructor C_i is

$$\begin{aligned}\tau_0@C_i \;=\;& NR_\alpha(\{\})(t_{i1}\ldots t_{ic(i)})\ (S_\alpha(t_{i1}\ldots t_{ic(i)})(t_0)(\tau_0)\ \alpha_i)\\ & \text{where } \tau_0 = (\alpha_1,\ldots,\alpha_n)\\ & \text{and } t_0 = C_1\ t_{11}\ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1}\ldots t_{nc(n)}\end{aligned}$$

Intuitively speaking, we first substitute τ_0 at every occurrence of t_0 in α_i with S_τ, then reduce the result with NR_α. It is clear that the result is a uniform application context over $t_{i1}\ldots t_{ic(i)}$. □

Definition 13 Let a data type $t_0 ::= C_1\ t_{11}\ldots t_{1c(1)} \mid \ldots \mid C_n\ t_{n1}\ldots t_{nc(n)}$ be given, and let $\alpha_1, \ldots, \alpha_n$ be uniform application contexts, $\alpha_i \in Actx(\{\})(t_{i1},\ldots,t_{ic(i)})$.

The *folding* over t_0 of $\alpha_1, \ldots, \alpha_n$, is defined by

$$\begin{aligned}t_0[\alpha_1,\ldots,\alpha_n] \;=\;& (NR_\tau(\{\})(t_0)\ \tau') \sqcup (\textstyle\bigsqcup \bigcup_{i=1}^{n} ST_\alpha(t_{i1}\ldots t_{ic(i)})(t_0)\ \alpha_i)\\ & \text{where } \tau' = (\alpha_1, \ldots, \alpha_n)\end{aligned}$$

Here $\sqcup$ and $\bigsqcup$ is least upper bound in $Eot(\{\})(t_0)$. Intuitively speaking, we construct a τ-tree τ' from the application contexts α_i; find all occurrences of t_0 in the α_i with ST_α; reduce (normalize) τ'; and finally we find the least upper bound of all the evaluation order types. □

Now let us see how the above definitions apply to the restricted case $t_0 = \texttt{nlist}$. Recall that a uniform eot over `nlist` has form $\tau_0 = (\alpha_1,\alpha_2)$ where $\alpha_1 = (a_1,())$ is the application context for `Nil`, and $\alpha_2 = (a_2,(\tau_b,\oplus))$ is the application context for `Cons`, a_1 is in $AEod(0)$, a_2 in $AEod(2)$, and τ_b in $Eot(\texttt{nat})$.

Example 6 Let τ_0 as specified above be given. The unfolding τ_0@`Nil` for `Nil` is

$$\begin{aligned}\tau_0@\texttt{Nil} &= NR_\alpha(\{\})()(S_\alpha()(\texttt{nlist})(\tau_0)\ \alpha_1)\\ &= NR_\alpha(\{\})()(a_1,())\\ &= (a_1,())\end{aligned}$$

as expected. The unfolding τ_0@`Cons` for `Cons` is

$$\begin{aligned}\tau_0@\texttt{Cons} &= NR_\alpha(\{\})(\texttt{nat nlist})(S_\alpha(\texttt{nat nlist})(\texttt{nlist})\ \alpha_2)\\ &= NR_\alpha(\{\})(\texttt{nat nlist})\\ &\quad\ (a_2,(S_\tau(\texttt{nat})(\texttt{nlist})(\tau_0)\ \tau_b,\ S_\tau(\texttt{nlist})(\texttt{nlist})(\tau_0)\ \oplus))\\ &= NR_\alpha(\{\})(\texttt{nat nlist})(a_2,(\tau_b,\tau_0))\\ &= (a_2,(NR_\tau(\{\})(\texttt{nat})\ \tau_b,\ NR_\tau(\{\})(\texttt{nlist})\ \tau_0))\\ &= (a_2,(\tau_b,\ (NR_\alpha\{\texttt{nlist}\}()\ \alpha_1,\ NR_\alpha\{\texttt{nlist}\}(\texttt{nat nlist})\ \alpha_2)))\\ &= (a_2,(\tau_b,\ ((a_1,()),\ (a_2,(\tau_b,\oplus)))))\\ &= (a_2,(\tau_b,\ \tau_0))\end{aligned}$$

as expected: the second argument to `Cons` is in the same context τ_0 as the `Cons` application itself. □

Example 7 Now consider folding of `nlist`. Let $\alpha_1 = (a_1,())$ and $\alpha_2 = (a_2,(\tau_{21},\tau_{22}))$ be application contexts for the branches `Nil => e`$_1$ and `Cons x`$_1$ `x`$_2$ `=> e`$_2$ of a `case` on `nlist`. Here τ_{21} and τ_{22} are the contexts of $\mathtt{x}_1$ and $\mathtt{x}_2$ in $\mathtt{e}_2$.

$$\begin{array}{rcll} \tau_{21} & = & \tau_{b1} & \in \mathit{Eot}(\{\})(\mathtt{nat}) \\ \tau_{22} & = & ((b_1,()),\ (b_2,(\tau_{b2},\oplus))) & \in \mathit{Eot}(\{\})(\mathtt{nlist}) \end{array}$$

By definition

$$\begin{array}{rcl} \mathtt{nlist}[\alpha_1,\ \alpha_2] & = & (NR_\tau(\{\})(\mathtt{nlist})(\alpha_1,\alpha_2)) \\ & & \sqcup\ (\bigsqcup\ (ST_\alpha()(\mathtt{nlist})\alpha_1 \cup ST_\alpha(\mathtt{nat\ nlist})(\mathtt{nlist})\alpha_2)) \end{array}$$

But

$$\begin{array}{l} NR_\tau(\{\})(\mathtt{nlist})(\alpha_1,\alpha_2) \\ \quad = (NR_\alpha\{\mathtt{nlist}\}()\alpha_1,\ NR_\alpha\{\mathtt{nlist}\}(\mathtt{nat\ nlist})\alpha_2) \\ \quad = ((a_1,()),\ (a_2,(NR_\tau\{\mathtt{nlist}\}(\mathtt{nat})\tau_{21},\ NR_\tau\{\mathtt{nlist}\}(\mathtt{nlist})\tau_{22}))) \\ \quad = ((a_1,()),\ (a_2,(\tau_{b1},\oplus))) \end{array}$$

and $ST_\alpha()(\mathtt{nlist})\ \alpha_1 = \{\}$ and

$$\begin{array}{l} ST_\alpha(\mathtt{nat\ nlist})(\mathtt{nlist})\ \alpha_2 \\ \ = ST_\tau(\mathtt{nat})(\mathtt{nlist})\ \tau_{21} \cup ST_\tau(\mathtt{nlist})(\mathtt{nlist})\ \tau_{22} \\ \ = \{\} \cup \{\ NE_\tau(\{\})(\mathtt{nlist})\tau_{22}\ \} \\ \ = \{\ \tau_{22}\ \} \end{array}$$

Thus $\mathtt{nlist}[\alpha_1,\ \alpha_2] = ((a_1,()),\ (a_2,(\tau_{b1},\oplus))) \sqcup \tau_{22} = ((a_1 \sqcup b_1,()),\ (a_2 \sqcup b_2,(\tau_{b1} \sqcup \tau_{b2},\oplus)))$.

This is as expected: the argument eod $a_1 \sqcup b_1$ for `Nil` is the least upper bound of its argument eod a_1 in $\mathtt{e}_1$ and the argument eod b_1 extracted from the context τ_{22} for $\mathtt{x}_2$ in $\mathtt{e}_2$, and similarly for the argument eod $a_2 \sqcup b_2$ for `Cons`. Also, the context $\tau_{b1} \sqcup \tau_{b2}$ of the `nat` argument to `Cons` is the least upper bound of the context of $\mathtt{x}_1$ in $\mathtt{e}_2$ with the context extracted from the context τ_{22} of $\mathtt{x}_2$ in $\mathtt{e}_2$. □

If anything is clear from these computations it is that the general treatment is an overkill for the `nlist` example. However, for recursively defined types with a slightly more complicated recursion pattern, this generality is needed. Consider for instance the contrived example

c ::= D d | E e
d ::= N `nat`
e ::= C c

For unfolding c@E, substitution (with S_α) and reduction (with NR_α) is necessary. For folding c$[\alpha_1,\alpha_2]$, reduction (with NR_τ), and locating and expanding the subtrees corresponding to c (with ST_α and NE_τ) is necessary. Thus this example exercises all the machinery introduced above.

6.10 Related work

The path analysis work by Bloss and Hudak has served as reference and inspiration [8] [6]. Bloss and Hudak's forwards analysis gives more precise information than our backwards analysis (for atomic data) but is more expensive. In contrast to our analysis, their analysis works for higher order languages but is restricted to atomic data, and it is not clear what kind of extensions are needed to make their analysis work for lazy data structures.

One may ask why this could not be achieved simply by encoding data structures as higher order functions and then applying Bloss and Hudak's higher order analysis, thus avoiding a whole new theory. Some answers may be found in Section 4.4.2. First, the complexity of higher order path analysis is superexponential in the depth of types in the expressions being analysed, and the encoding of data structures as functions give very complex types. Secondly, it may be that for type reasons, higher order path analysis cannot be applied at all to recursive functions involving data structures encoded as functions [35].

The only other automatic evaluation order analysis we know of is that of Draghicescu and Purushothaman [17] [18]. This is still restricted to first order languages without lazy data structures, but their analysis can accommodate other evaluation strategies than pure laziness. In particular those which make use of strictness information (only) to move evaluation of arguments earlier.

The original motivation for this study stems from a desire to generalize a technique for globalization of function parameters to lazy languages [31], which is similar to update analysis [7]. It is not yet clear whether the results obtained here are actually useful in that connection.

The whole framework obtained in this chapter can be seen as a special instance of the general backwards analysis theory of Hughes [38].

Backwards strictness analysis of general data structures by abstract interpretation of continuations was done by Hughes [36] [37]. Another account, based on projections, was given by Wadler and Hughes [73].

Chapter 7

Globalization for Partial Applications

This chapter describes an analysis to detect when a function parameter can be replaced by a global variable, and a transformation called globalization which does the replacement. The analysis works for an untyped, strict, higher order functional language, and extends previous work by Schmidt, Sestoft, and Fradet [57] [59] [24] in that also variables which can be captured in closures may be globalized. This chapter is a slightly edited version of a published paper which presents joint work with Carsten K. Gomard [31]. Much of the text is likely to appear also in his forthcoming Ph.D. thesis [29].

In this chapter we first present a strict example language, a sequentialized variant of it, and its operational semantics (Section 7.2). After this we discuss the subject proper: liveness and global liveness (Section 7.3), interference (Section 7.4), variable groups (Section 7.5), globalization (Section 7.6), and correctness (Section 7.7). Finally we describe some experiments and related work, and give a conclusion.

7.1 Introduction

Straightforward implementations of strict functional languages may be quite slow and consume much storage. One inefficiency problem is that even essentially global data structures must be passed around as parameters to functions, causing copying of data structures or of pointers to them. Therefore it is an interesting problem to detect automatically such essentially global parameters in functional programs and replace them by global variables which are assigned values in an imperative manner.

This is particularly useful when one attempts to derive language implementations directly from denotational or functional specifications. If one could recognize the parameters representing the run-time store in a functional specification of, say, Pascal, then this would allow a much more natural and efficient Pascal implementation to be derived from the specification.

7.1.1 Previous work

This was precisely the goal of Schmidt's work [57]. He called this desirable property *single-threading*, and showed that single-threading criteria could be formulated elegantly as conditions on the types assigned to all (sub)expressions in the language specification.

A different approach to replacing function parameters by global variables was taken by Sestoft, who called this process globalization [58]. Conditions for globalizability were formulated in terms of definition-use paths (a kind of abstract traces of the evaluation of a program), thus making the dependency on order of evaluation explicit. This method also worked for untyped programs and was sometimes able to detect more potential global variables than Schmidt's. A price for the extra power was more complicated algorithms. Where Schmidt did not make any assumptions about evaluation order (except for strictness), Sestoft imposed a fixed evaluation order.

Fradet formulated single-threading criteria as syntactic conditions on typed expressions transformed into continuation-passing style [24]. Fradet also imposed a fixed evaluation order.

7.1.2 The present work

All the above-mentioned techniques were unable to detect globalizability of variables that might be captured in a *closure* or *partial function application*. To solve this problem we propose yet a new method in this chapter. The method is based on a (suitably extended) concept of live variable [2], and the observation that non-live instances can be overwritten without any adverse effect on the future computation. If at every creation of a new instance of `x` there is no live instance of it, then there will ever only be one instance which can therefore be allocated globally. Instead of creating a new instance, one can overwrite this global instance without affecting the future computation.

To understand the concept of *instance*, assume that every function application ($\mathtt{e}_1$ $\mathtt{e}_2$) extends a closure (initially possibly empty) with the value of $\mathtt{e}_2$. The value of $\mathtt{e}_2$ becomes the value of the first parameter $\mathtt{x}^i_j$ of the function to which $\mathtt{e}_1$ evaluates. We then say that evaluation of this application results in the creation of a new instance of $\mathtt{x}^i_j$, the one that holds the value of $\mathtt{e}_2$.

In addition to globalizing single variables, we are able to find *groups* of variables which can be allocated as the same global variable. One example of such a group is the set of variables holding the store value in a functional interpreter (specification) for an imperative language. For this we need to know not only whether there are live instances of `x` when a new instance of `x` is created, but also which other variables have live instances at that point.

The creation of a new instance of variable `x` while some instance of variable `z` is live is called interference, because it (evidently) prevents `z` and `x` from being globalized together, that is, replaced by the same global variable. This piece of information is represented by the interference pair (`z`,`x`).

Using the concept of interference, the condition for globalization can be stated as follows: a variable that never interferes with itself can be allocated as a global variable.

The extension of this condition to groups of variables is: A set of variables which do not interfere with each other can be allocated as one and the same global variable.

We give an interference analysis which finds the set of interference pairs in a given program (under call-by-value evaluation with left to right evaluation of function arguments), and an algorithm which groups globalizable variables using the information from the interference analysis.

Example 8 Consider the program

```
map f l = if   (null? l)
          then ()
          else (cons (f (car l)) (map f (cdr l)))
add n x = n + x
goal  i = append (map (add 2) i) (map (add 3) i))
```

Our algorithms find that *all* function parameters may be replaced by global variables but that no two variables may be grouped, *i.e.*, replaced by the same global variable. It finds that the first variable `n` of `add` is globalizable although it may be captured in a closure. Previous analyses, in particular those of Schmidt, Sestoft, and Fradet mentioned above, would not have discovered this fact. □

This work started with an attempt to extend Sestoft's previous method of definition-use paths and path semantics (an instrumented operational semantics) [58]. However, we failed to find a simple way to do that, and turned instead to a special syntax for making evaluation order explicit in an expression. In that respect this work resembles that of Fradet, who uses continuation expressions for sequencing [24] where we use so-called sequentialized expressions (explained below). Like Sestoft's earlier method, but unlike Fradet's, the present method uses a closure analysis to handle higher order functions, and thus achieves better precision than Fradet's.

7.2 The strict language H

The language we analyse is a higher order functional language with call-by-value evaluation and left-to-right evaluation of subexpressions. We call this language H for historical reasons. An H program is a list of recursive function definitions without local function definitions, together with a goal expression $\mathtt{e}_0$. A functional program can always be brought into this form by lambda-lifting [42]:

$$
\begin{array}{ll}
\mathtt{letrec} & \mathtt{f}^1\ \mathtt{x}^1_1 \ldots \mathtt{x}^1_{arity_1} \;=\; \mathtt{e}^1 \\
& \ldots \\
& \mathtt{f}^n\ \mathtt{x}^n_1 \ldots \mathtt{x}^n_{arity_n} \;=\; \mathtt{e}^n \\
\mathtt{in}\ \mathtt{e}_0 &
\end{array}
$$

The $\mathtt{f}^i$ are called *defined functions*, the $\mathtt{e}^i$ are called *function bodies*, and $arity_i = arity(\mathtt{f}^i)$ is the *arity* of $\mathtt{f}^i$. The arity must be non-zero for all functions $\mathtt{f}^i$. Input to the program is through the free variables of the goal expression $\mathtt{e}_0$.

7.2.1 Source expressions

The body expressions $\mathtt{e}^i$ have form:

e ::=	x	Function parameter
|	$\mathtt{f}^i$	Defined function
|	if e e e	Conditional
|	A(e,...,e)	Basic function
|	(e e)	Application

Constants are treated as niladic (argumentless) base functions.

7.2.2 Operational semantics and the SECD machine

The operational semantics of our language is given via a simple translation to code for the SECD-machine [44]. We believe this is a reasonable choice of machine since Landin invented it to formalize the mechanical evaluation of call by value lambda calculus. Another possibility would be the Categorical Abstract Machine [12].

First we give the details of our variant of the SECD machine, then we describe the meaning of the instructions by state transition rules in Figure 2.

There are three kinds of values handled in the semantics. The first kind is a base value (or, atomic value) b such as 3 or true. The second kind of value is a *closure*, or *partial application*. A closure $\mathtt{f}^i[v_1,\ldots,v_m]$ is a defined function symbol $\mathtt{f}^i$ together with values $v_j \in \mathit{Val}$ for some but not all of $\mathtt{f}^i$'s parameters. That is, we must have $0 \leq m < \mathit{arity}(\mathtt{f}^i)$. A closure is the only kind of functional value in H. The third kind of value is a *data structure value*, which has form $C[v_1,\ldots,v_m]$, where $C \in \mathit{Cnstr}$ is a *constructor* of arity $m \geq 0$, and the v_j's are values. More formally:

$$\begin{array}{lll}
\mathtt{f}^i : \mathit{Fun} & & \text{Defined functions} \\
b : \mathit{BVal} & & \text{Base values} \\
v : \mathit{Val} & = & \mathit{BVal} + \mathit{Clo} + \mathit{Struc} \\
\quad \mathit{Clo} & = & \{\ \mathtt{f}^i[v_1,\ldots,v_m] \mid \mathtt{f}^i \in \mathit{Fun},\ v_j \in \mathit{Val},\ 0 \leq m < \mathit{arity}(\mathtt{f}^i)\ \} \\
\quad \mathit{Struc} & = & \{\ C[v_1,\ldots,v_m] \mid C \in \mathit{Cnstr},\ v_j \in \mathit{Val},\ m = \mathit{arity}(C)\ \} \\
\mathtt{x} : \mathit{Param} & & \text{Function parameters} \\
\mathtt{t} : \mathit{TVar} & & \text{Temporary variables} \\
\quad \mathit{Var} & = & \mathit{Param} + \mathit{TVar}
\end{array}$$

The state $\sigma : \mathit{State} = \mathit{Code} \times \mathit{Stack} \times \mathit{Env} \times \mathit{Dump}$ of the abstract machine has the following four components:

$$\begin{array}{llll}
\mathrm{C} : \mathit{Code} & & & \text{the control} \\
\mathrm{S} : \mathit{Stack} & = & \mathit{Val}^* & \text{the stack} \\
\mathrm{E} : \mathit{Env} & = & \mathit{Param} \to \mathit{Val} & \text{the environment} \\
\mathrm{D} : \mathit{Dump} & = & (\mathit{Code} \times \mathit{Stack} \times \mathit{Env})^* & \text{the dump}
\end{array}$$

A piece of code is a sequence of instructions. The instructions of our variant of the machine are listed below.

`Ret`	v:S	E	(C′,S′,E′):D	⇒	C′	v:S′	E′	D	
`Var x`; C	S	E	D	⇒	C	(E `x`):S	E	D	
`Bas A`; C	v_m:..:v_1:S	E	D	⇒	C	(`A` v_1 … v_m):S	E	D	
`Fun` $\mathtt{f}^i$; C	S	E	D	⇒	C	$\mathtt{f}^i$[]:S	E	D	
`If` $C_1 C_2$; C	true:S	E	D	⇒	C_1; C	S	E	D	
`If` $C_1 C_2$; C	false:S	E	D	⇒	C_2; C	S	E	D	
`App`; C	v_{m+1}:$\mathtt{f}^i[v_1,\ldots,v_m]$:S	E	D	⇒	C	$\mathtt{f}^i[v_1,\ldots,v_{m+1}]$:S	E	D	if $m+1 < arity_i$
`App`; C	v_{m+1}:$\mathtt{f}^i[v_1,\ldots,v_m]$:S	E	D	⇒	C_i	[]	$[\mathtt{x}^i_j \mapsto v^i_j]$	(C,S,E):D	if $m+1 = arity_i$

Here C_i is the code for the body of defined function $\mathtt{f}^i$; and [] is the empty stack.

Figure 2: Basic SECD Machine Rules

$$
\begin{array}{lll}
Code & = & Ins^* \\
Ins & ::= & \texttt{Ret} \qquad\qquad \text{Return or halt} \\
 & | & \texttt{Var x} \qquad\quad \text{Function parameter} \\
 & | & \texttt{Bas A} \qquad\quad \text{Basic function} \\
 & | & \texttt{Fun f}^i \qquad\ \text{Defined function} \\
 & | & \texttt{If}\ Code\ Code \quad \text{Conditional} \\
 & | & \texttt{App} \qquad\qquad \text{Application}
\end{array}
$$

The evaluation rules of the basic SECD machine are shown in Figure 2. The scheme for compilation of source expressions into SECD instructions is the natural one:

$$
\begin{array}{ll}
\mathcal{C}[\![\, \mathtt{x} \,]\!] & = \mathtt{Var\ x} \\
\mathcal{C}[\![\, \mathtt{f}^i \,]\!] & = \mathtt{Fun\ f}^i \\
\mathcal{C}[\![\, \mathtt{if\ e_1\ e_2\ e_3} \,]\!] & = \mathcal{C}[\![\, \mathtt{e}_1 \,]\!] \ ;\ \mathtt{If}\ \mathcal{C}[\![\, \mathtt{e}_2 \,]\!]\ \mathcal{C}[\![\, \mathtt{e}_3 \,]\!] \\
\mathcal{C}[\![\, \mathtt{A\ e_1\ \ldots\ e}_n \,]\!] & = \mathcal{C}[\![\, \mathtt{e}_1 \,]\!] \ ;\ \ldots\ ;\ \mathcal{C}[\![\, \mathtt{e}_n \,]\!] \ ;\ \mathtt{Bas\ A} \\
\mathcal{C}[\![\, \mathtt{e_1\ e_2} \,]\!] & = \mathcal{C}[\![\, \mathtt{e}_1 \,]\!] \ ;\ \mathcal{C}[\![\, \mathtt{e}_2 \,]\!] \ ;\ \mathtt{App}
\end{array}
$$

The operator "`;`" appends two sequences of instructions, and appends to both code sequences in `If`, so (`If` C_1 C_2)`;`C_3 equals `If` (C_1`;`C_3) (C_2`;`C_3). This does not give ordinary linear machine code with labels and jumps, but this is immaterial for our purposes.

The code generated for a defined function $\mathtt{f}^i$ is $C_i = \mathcal{C}[\![\, \mathtt{e}^i \,]\!]$`;Ret`, where $\mathtt{e}^i$ is the body of $\mathtt{f}^i$.

7.2.3 Sequentialized expressions

To make the order of evaluation of subexpressions explicit on the expression level, we shall use an alternative syntax, called *sequentialized expressions*. This simplifies the liveness and interference analyses.

Sequentialized expressions have form `sexp` where

$$
\begin{array}{lll}
\mathtt{sexp} & ::= & \mathtt{texp} \qquad\qquad\qquad\qquad \text{Sequential expression} \\
 & | & \mathtt{let\ t\ =\ sexp\ in\ sexp} \\
 & | & \mathtt{if\ t\ sexp\ sexp} \\
\mathtt{texp} & ::= & \mathtt{x} \qquad\qquad\qquad\qquad\quad\ \text{Trivial expression} \\
 & | & \mathtt{f}^i \\
 & | & \mathtt{A(t_1,\ldots,t}_n\mathtt{)} \\
 & | & \mathtt{(t_1\ t_2)}
\end{array}
$$

Note the restrictions: in applications, all subexpressions must be `let`-bound variables `t`. Where two `sexp` subexpressions may occur, one must be evaluated wholly before the other (as in `let`), or they must be alternatives (as in `if`). The `t` variables are "fresh", all distinct from the ordinary variables `x`.

Thus sequentialized form is a kind of "functional three-address code" and serves (at least) one of the purposes of classical three-address code: To make liveness and order of evaluation explicit [2]. (Another potential application is a form of source-level register allocation.)

7.2.4 Sequentialization

The translation scheme from source expressions to sequentialized expressions is very straightforward, and it is quite obvious that sequentialization preserves not only the meaning but also the order of evaluation of subexpressions.

$$
\begin{array}{ll}
\mathcal{S}[\![\,\mathtt{x}\,]\!] & = \mathtt{x} \\
\mathcal{S}[\![\,\mathtt{f}^i\,]\!] & = \mathtt{f}^i \\
\mathcal{S}[\![\,\mathtt{if}\ \mathtt{e}_1\ \mathtt{e}_2\ \mathtt{e}_3\,]\!] & = \mathtt{let}\ \mathtt{t}_1\ \mathtt{=}\ \mathcal{S}[\![\,\mathtt{e}_1\,]\!] \\
 & \ \mathtt{in}\ \ \mathtt{if}\ \mathtt{t}_1\ \mathcal{S}[\![\,\mathtt{e}_2\,]\!]\ \mathcal{S}[\![\,\mathtt{e}_3\,]\!] \\
\mathcal{S}[\![\,\mathtt{A}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_n\,]\!] & = \mathtt{let}\ \mathtt{t}_1\ \mathtt{=}\ \mathcal{S}[\![\,\mathtt{e}_1\,]\!]\ \mathtt{in} \\
 & \ \ldots \\
 & \ \mathtt{let}\ \mathtt{t}_n\ \mathtt{=}\ \mathcal{S}[\![\,\mathtt{e}_n\,]\!] \\
 & \ \mathtt{in}\ \ \mathtt{A}\ \mathtt{t}_1\ldots\ \mathtt{t}_n \\
\mathcal{S}[\![\,\mathtt{e}_0\ \mathtt{e}_1\,]\!] & = \mathtt{let}\ \mathtt{t}_1\ \mathtt{=}\ \mathcal{S}[\![\,\mathtt{e}_1\,]\!]\ \mathtt{in} \\
 & \ \mathtt{let}\ \mathtt{t}_2\ \mathtt{=}\ \mathcal{S}[\![\,\mathtt{e}_2\,]\!] \\
 & \ \mathtt{in}\ \ (\mathtt{t}_1\ \mathtt{t}_2)
\end{array}
$$

The scheme for compiling sequentialized expressions to SECD-machine code is trivial because of the particular way in which sequential expressions are generated:

$$
\begin{array}{ll}
\mathcal{C}_{\mathcal{S}}[\![\,\mathtt{x}\,]\!] & = \mathtt{Var}\ \mathtt{x} \\
\mathcal{C}_{\mathcal{S}}[\![\,\mathtt{f}^i\,]\!] & = \mathtt{Fun}\ \mathtt{f}^i \\
\mathcal{C}_{\mathcal{S}}[\![\,\mathtt{if}\ \mathtt{t}_1\ \mathtt{e}_2\ \mathtt{e}_3\,]\!] & = \mathtt{If}\ \mathcal{C}_{\mathcal{S}}[\![\,\mathtt{e}_2\,]\!]\ \mathcal{C}_{\mathcal{S}}[\![\,\mathtt{e}_3\,]\!] \\
\mathcal{C}_{\mathcal{S}}[\![\,\mathtt{A}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n\,]\!] & = \mathtt{Bas}\ \mathtt{A} \\
\mathcal{C}_{\mathcal{S}}[\![\,\mathtt{t}_1\ \mathtt{t}_2\,]\!] & = \mathtt{App} \\
\mathcal{C}_{\mathcal{S}}[\![\,\mathtt{let}\ \mathtt{t}_1\ \mathtt{=}\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}\,]\!] & = \mathcal{C}_{\mathcal{S}}[\![\,\mathtt{e}_1\,]\!]\ ;\ \mathcal{C}_{\mathcal{S}}[\![\,\mathtt{e}\,]\!]
\end{array}
$$

In terms of the operational semantics, `let`-bound variables in sequentialized expressions correspond to positions on the evaluation stack S. Also, the effect of evaluating the binding $\mathtt{t}_1\ \mathtt{=}\ \mathtt{e}_1$ in a `let`-expression is to push the value of $\mathtt{e}_1$ onto S.

It is easy to prove that exactly the same SECD machine code is generated for a sequentialized expression as for its source expression, *i.e.*, that $\mathcal{C}_{\mathcal{S}} \circ \mathcal{S} = \mathcal{C}$. This ensures that although liveness and interference analysis is done on sequentialized expressions, the analysis results may be used on the original expressions.

7.2.5 Closure analysis for the H language

Our method for doing liveness and interference analysis of a program requires a preceding closure analysis phase. The strict H language used in this chapter has a different syntax and semantics than the lazy L language analysed by the closure analysis of Chapter 4. However, there is no essential difference between closure analysis of a strict language and closure analysis of a non-strict one, since the results describe only the possible data flow, not the order of evaluation or the definedness of expressions.

Here we shall not define but just assume there a *closure analysis* ca for H. In the closure analysis for H a concrete closure $\mathtt{f}^i[v_1,\ldots,v_m]$ is represented by the *abstract closure* $(\mathtt{f}^i,m)$:

$$(\mathtt{f}^i,m) \in \mathit{AClo} = \{ (\mathtt{f}^i,m) \mid 0 \leq m < \mathit{arity}(\mathtt{f}^i) \}$$

We shall use the symbol $\mathrm{ca}[\![\,\mathtt{e}\,]\!]$ for a set of abstract closures, and require that for every concrete closure $\mathtt{f}^i[v_1,\ldots,v_m]$ that may be part of a possible value of e, the corresponding abstract closure $(\mathtt{f}^i,m)$ is in $\mathrm{ca}[\![\,\mathtt{e}\,]\!]$.

$$\begin{array}{lcl} \mathrm{ca} & : & \mathit{Expr} \rightarrow \wp(\mathit{AClo}) \\ \mathrm{ca}[\![\,\mathtt{e}\,]\!] & \supseteq & \{ (\mathtt{f}^i,m) \mid v \in \mathit{Val} \text{ is a possible value of } \mathtt{e}, \\ & & \text{and } \mathtt{f}^i[v_1,\ldots,v_m] \in \mathrm{fun}(v) \} \end{array}$$

The value $v \in \mathit{Val}$ is a possible value of e if there is any evaluation in which e evaluates to v in the operational semantics of the language.

(Loosely speaking, an H closure $\mathtt{f}^i[v_1,\ldots,v_m]$ corresponds to an L closure $\lambda^\ell\mathtt{x}.\mathtt{e}\bullet\mathrm{E}$ where $\mathrm{E} = [v_m,\ldots,v_1]$. In the analysis the abstract closure $(\mathtt{f}^i,m)$ in H corresponds to the lambda label ℓ in L.)

Note that for data structures the closure analysis for H is quite different from that for L. In L a data structure would be encoded as a lambda expression, and closure analysis of an expression which evaluates to a data structure would just return the label of the lambda representing the topmost constructor. In the closure analysis for H we are interested in all the functions possibly held also *inside* the data structure.

The closure analysis for H (except data structures) is described in detail and proved correct in [58, Sect. 4.3].

7.3 Live variables and globally live variables

A variable x is *locally live* at a program point if in the present scope (*i.e.*, function body) there is a temporally later possible use of it with no intervening `let`-definition of the variable. This definition is dependent on a notion of time and thus on operational aspects such as the order of evaluation of subexpressions.

Example 9 In the body of `f x y z = x + y + z`, all of `x`, `y`, and `z` are locally live before evaluation of `x`; variables `y` and `z` are live just before evaluation of `y`; `z` is live just before evaluation of `z`; and no variable is locally live after evaluation of the expression. □

The concepts defined here are *approximate*. That is, they concern classes of computations, not individual computations on ordinary values. So we say that variable `x` is live after expression `e` in `if e then x else 7` because there is a *possible* later use of `x`, even though in a particular computation there may not be such a later use.

A program point in an expression is a point in the instruction stream compiled for the expression. In terms of the compiled code, a function parameter `x` is locally live at a program point if the instruction (`Var x`) occurs in the code after that point.

7.3.1 The use of sequentialized expressions

This latter definition of local liveness does not take the `let`-bound temporary variables `t` into account. The `t` variables correspond to stack positions and are not mentioned in the compiled code. We shall use the `t` variables precisely to make apparent already at the expression level which temporary results exist on the computation stack: those bound to live `t` variables.

In the example program below all subexpressions are superscripted with a label. The expressions are evaluated in the order 2, 3, 1, 5, 6, 4, 0. We see that the value `h`[2] is pushed on the evaluation stack S at time 1 and popped at time 0:

$$
\begin{array}{lll}
\texttt{k y} & \texttt{=} & \texttt{(cons (h}^2\ \texttt{2}^3\texttt{)}^1\ \texttt{(g}^5\ \texttt{3}^6\texttt{)}^4\texttt{)}^0 \\
\texttt{h z w} & \texttt{=} & \texttt{...} \\
\texttt{g x} & \texttt{=} & \texttt{...}
\end{array}
$$

Example 10 In the sequentialized form of that program,

```
k y = let t1 = let t2 = h in
               let t3 = 2
               in (t2 t3)
      in
          t4 = let t5 = g in
               let t6 = 3
               in (t5 t6)
      in  (cons t1 t4)
h z w = ...
g x   = ...
```

it is obvious that `t1` is locally live, and hence that the temporary value `h`[2] bound to `t1` exists on the evaluation stack, during the definition of `t4`. Note that `t1` and hence the closure `h`[2] ceases to be live after the use to `t1`, corresponding to the popping at time 0 mentioned above. □

While ordinary (local) liveness of function parameters is a "backwards" concept, the liveness of temporary results is a kind of "forwards" concept. We can now consider sequentialized programs as a way to turn forwards liveness of temporary *results* on the stack S into ordinary backwards liveness of temporary *variables*. The advantage of this is that the analysis becomes simpler to express.

7.3.2 Non-local liveness

In this section we show two kinds of non-local liveness: indirect liveness and global liveness.

In the sequentialized program shown in Example 10 above, a closure `h`[2] is constructed and is held in the locally live variable `t1` while the right hand side of `t4` is being evaluated. Thus although `h`'s first parameter `z` cannot appear in the body of `k`, and thus is not *locally* live, during evaluation of `t4` an instance of `z` exists in a closure on the stack S. We shall say below that `z` is indirectly live because there is an instance of it in the closure `h`[2], and hence a possible later use of `z` in a full application of `h`.

Operationally, there will be a reference to `h`[2] (namely, the value of `t1`) on the computation stack S while `(g 3)` is computed. Since `h`[2] in turn holds a value (namely, 2) for `z`, `z` is indirectly live while `(g 3)` is being evaluated.

Example 11 Consider the program

```
f1 z = (f2 3) + z
f2 x = ...
```

Here `z` is locally live in the body of `f1` just after the call (full application) of `f2`. Hence when evaluating the body of `f2` there is a later use of the non-local variable `z`, which is therefore live everywhere in the body of `f2` although it does not appear in `f2`. We shall say below that `z` is globally live in `f2`. Global liveness is further discussed in Section 7.3.4 below.

Operationally, variable `z` occurs in the code C′ on the dump D = (C′,S′,E′) : D′ while evaluating the body of `f2`. Thus `z` is globally live while evaluating the body of `f2`. □

7.3.3 Indirect liveness and capture of variables

We say that a value v may *capture* a set of variables, namely, those which have an instance in a closure in v. We can now define that a variable is (locally) *indirectly live* at a program point if it may be captured in the value of some locally live variable.

A base (atomic) value b captures no variables. A closure value $\mathtt{f}^i[v_1,\ldots,v_m]$ captures the first m parameters $\mathtt{x}^i_1$, ..., $\mathtt{x}^i_m$ of $\mathtt{f}^i$, and those parameters that are captured in the values $v_1,\ldots,v_m$. A data structure value $C[v_1,\ldots,v_m]$ captures the union of the sets captured by its component values $v_1,\ldots,v_m$. Capture extends linearly to sets of values: The set of parameters captured by a set of values is the union of the captured sets. The exact capture analysis *cap* computes the set of parameters captured in a set of values. (It ignores the temporary `t` variables since these are not interesting outside the function definitions they occur in.)

$$cap : \wp(Val) \rightarrow \wp(Param)$$

$$\begin{aligned}
cap(\{b\}) &= \{\} \\
cap(\{\mathtt{f}^i[v_1,\ldots,v_m]\}) &= \{\ \mathtt{x}^i_1,\ \ldots,\ \mathtt{x}^i_m\ \} \\
&\quad \cup\ cap(\{v_1,\ldots,v_m\}) \\
cap(\{C[v_1,\ldots,v_m]\}) &= cap(\{v_1,\ldots,v_m\}) \\
cap(V_1 \cup V_2) &= cap(V_1) \cup cap(V_2)
\end{aligned}$$

In terms of the operational semantics, a variable `x` is (locally) indirectly live at a program point if for some possible machine state (C,S,E,D) where C corresponds to that point, either `x` is captured in the value (E `y`) of some locally live variable `y`, or `x` is captured in some value v in the evaluation stack S. The latter case corresponds to some `let`-bound variable `t` being locally live, and `x` being captured in its value.

For use in the liveness and interference analyses we need an approximate capture analysis. Such a function *acap* can be defined using the closure analysis from Section 7.2.5:

$$
\begin{aligned}
&\mathit{acap}\colon \wp(\mathit{AClo}) \to \wp(\mathit{Param})\\
&\mathit{acap}(\{\}) &&= \{\}\\
&\mathit{acap}(V_1 \cup V_2) &&= \mathit{acap}(V_1) \cup \mathit{acap}(V_2)\\
&\mathit{acap}(\{(\mathtt{f}^i, m)\}) &&= \{\ \mathtt{x}_1^i, \ldots, \mathtt{x}_m^i\ \}\\
& &&\quad \cup\ \mathit{acap}(\mathrm{ca}[\![\,\mathtt{x}_1^i\,]\!] \cup \ldots \cup \mathrm{ca}[\![\,\mathtt{x}_m^i\,]\!])
\end{aligned}
$$

In the approximate liveness analysis we need to find all the variables which may be indirectly live due to a certain variable being (locally) live. This is called the *capture closure* of the variable. The capture closure $\mathit{ccl}(\mathtt{x})$ of a function parameter `x` is `x` itself together with the set of variables captured in the set of values that `x` may take on in any computation. The capture closure of a temporary variable `t` is the same except it does not include `t` itself: we are not (here) interested in the global liveness of `let`-bound variables.

$$
\begin{aligned}
&\mathit{ccl} : \wp(\mathit{Var}) \to \wp(\mathit{Param})\\
&\mathit{ccl}(\{\mathtt{x}\}) &&= \{\mathtt{x}\} \cup \mathit{acap}(\mathrm{ca}[\![\,\mathtt{x}\,]\!])\\
&\mathit{ccl}(\{\mathtt{t}\}) &&= \mathit{acap}(\mathrm{ca}[\![\,\mathtt{t}\,]\!])\\
&\mathit{ccl}(V_1 \cup V_2) &&= \mathit{ccl}(V_1) \cup \mathit{ccl}(V_2)
\end{aligned}
$$

Note that while capture is an exact concept, capture closure and *ccl* are approximate: they talk about all possible values of a variable, in all possible executions of the program, and are defined in terms of the approximate closure analysis.

7.3.4 Global liveness

We define that a variable `x` is *globally live* at a program point in the body $\mathtt{e}^i$ of function $\mathtt{f}^i$

1. if `x` is locally (possibly indirectly) live at that point,
2. or if `x` is globally live after some application ($\mathtt{e}_1$ $\mathtt{e}_2$) which can be a full application of $\mathtt{f}^i$. That is, if $\mathtt{e}_1$ can evaluate to the closure $\mathtt{f}^i[v_1, \ldots, v_{arity_i - 1}]$ in some computation, and `x` is globally live after ($\mathtt{e}_1$ $\mathtt{e}_2$).

The idea is that a variable is globally live if it may have a later use in the dynamically surrounding computation, *not* necessarily in the the lexically enclosing expression. This means that `x` may indeed be globally live in a completely different function than the one of which it is a parameter.

Explained in terms of the operational semantics, a variable `x` is globally live at a program point if it is locally indirectly live at that point, or there is a possible machine state (C,S,E,D) where the program point is in C, and D = (C′,S′,E′):D′, and `x` is globally live in (C′,S′,E′,D′).

Example 12 Example 11 gave one example of global liveness. For another, consider again Example 10 above. Here the parameter `z` of `h` is globally live everywhere in `g`'s body, because `z` is indirectly (locally) live just after the call to `g` in `k`'s body. □

For the rest of the chapter, we take *live* to mean globally (possibly indirectly) live. Recall that this includes also locally (possibly indirectly) live variables.

7.3.5 Computing liveness information

For each function $\mathtt{f}^i$ we compute the *avoid set* $\theta(\mathtt{f}^i)$, which is the set of variables that may be live while the function's body $\mathtt{e}^i$ is being evaluated. In Example 10 above, the avoid sets are as follows: $\theta(\mathtt{h}) = \{\}$, $\theta(\mathtt{k}) = \{\}$, $\theta(\mathtt{g}) = \{\mathtt{z}\}$. The *liveness analysis function* $\mathcal{L}$ computing this set for each $\mathtt{f}^i$ is given in Figure 3. Suppose

$$\mathcal{L}[\![\,\mathtt{e}\,]\!]\ \delta = (\delta',\ \theta')$$

If δ is the set of variables live *after* evaluation of `e`, then δ' is the set of variables live *before* evaluation of `e`. The other result component θ' is the contribution of `e` to the avoid sets of all functions in the program.

The most interesting rule in the definition of $\mathcal{L}$ is that for an application ($\mathtt{t}_1$ $\mathtt{t}_2$).

The first component of the result is the set of variables live before the application; this evidently is the set δ of those live after, plus $\mathtt{t}_1$ and $\mathtt{t}_2$.

The second component of the result is the contribution to the avoid sets θ. There can be a contribution only to those functions $\mathtt{f}^i$ which can have a full application at this expression. Those are the functions for which $\mathtt{t}_1$ may evaluate to $\mathtt{f}^i[v_1,\ldots,v_{arity_i-1}]$ for some values v_j. This set is approximated using the closure analysis by the set of functions for which ($\mathtt{f}^i$,$arity_i$-1) is a possible abstract closure value of $\mathtt{t}_1$. For all such functions $\mathtt{f}^i$, the contribution to the avoid set for $\mathtt{f}^i$ is the set of variables live after the application, that is, the capture closure $ccl(\delta)$ of δ.

Thus only full applications contribute to the avoid set, and the contribution to the avoid sets of the called functions is the set of variables live *after* the application.

A safe approximation to the set of globally live variables at a point in function $\mathtt{f}^i$ is found by computing the avoid set for function $\mathtt{f}^i$ and adding to it the locally (possibly indirectly) live variables.

7.4 Interference and interference analysis

Interference is the phenomenon that defining one variable `x` might adversely affect another variable `z` if `x` and `z` were allocated as the same global variable. We say that `x` *interferes*

$\mathtt{e}$: *Expr*	Sequentialized expressions (Section 7.2.3)
δ : $VarSet = \wp(\mathrm{Var})$	Set of live variables
θ : $GLEnv = Fun \rightarrow \wp(Param)$	The avoid set map
$\quad\perp_{GLEnv} = [\mathtt{f}^i \mapsto \{\}]$	The least avoid set map
ca : $ClMap = Expr \rightarrow \wp(AClo)$	The closure analysis (Section 7.2.5)
ccl: $VarSet \rightarrow \wp(AClo)$	Capture closure (Section 7.3.3)

$\mathcal{L}$: $Expr \rightarrow\ VarSet \rightarrow (\ VarSet \times\ GLEnv)$

$$
\begin{array}{ll}
\mathcal{L}[\![\mathtt{x}_j^i]\!]\ \delta & = (\{\mathtt{x}_j^i\} \cup \delta,\ \perp_{GLEnv}) \\
\mathcal{L}[\![\mathtt{f}^i]\!]\ \delta & = (\delta,\ \perp_{GLEnv}) \\
\mathcal{L}[\![\mathtt{if\ t\ e_1\ e_2}]\!]\ \delta & = (\{\mathtt{t}\} \cup \delta_1 \cup \delta_2,\ \theta_1 \sqcup \theta_2) \\
& \quad \text{where } (\delta_1,\ \theta_1) = \mathcal{L}[\![\mathtt{e}_1]\!]\ \delta \\
& \quad\qquad\quad (\delta_2,\ \theta_2) = \mathcal{L}[\![\mathtt{e}_2]\!]\ \delta \\
\mathcal{L}[\![\mathtt{A(t_1,\ldots,t_n)}]\!]\ \delta & = (\{\mathtt{t}_1\ \ldots\ \mathtt{t}_n\} \cup \delta,\ \perp_{GLEnv}) \\
\mathcal{L}[\![\mathtt{let\ t_1=e_1\ in\ e}]\!]\ \delta & = (\delta_1,\ \theta_0 \sqcup \theta_1) \\
& \quad \text{where } (\delta_0,\ \theta_0) = \mathcal{L}[\![\mathtt{e}]\!]\ \delta \\
& \quad\qquad\quad (\delta_1,\ \theta_1) = \mathcal{L}[\![\mathtt{e}_1]\!]\ (\delta_0 \backslash \{\mathtt{t}_1\}) \\
\mathcal{L}[\![\mathtt{t_1\ t_2}]\!]\ \delta & = (\delta \cup \{\mathtt{t}_1,\mathtt{t}_2\}, \bigsqcup \{\ [\mathtt{f}^i \mapsto ccl(\delta)] \mid (\mathtt{f}^i,\ arity_i{-}1) \in \mathrm{ca}[\![\mathtt{t}_1]\!]\ \})
\end{array}
$$

The desired result of the liveness analysis is the (pointwise inclusion-) least solution $\theta \in GLEnv$ to

$$\theta = \bigsqcup \{\ \mathrm{snd}(\mathcal{L}[\![\mathtt{e}^i]\!]\ (\theta\ \mathtt{f}^i)) \mid \mathtt{f}^i \text{ is a function }\}$$

Figure 3: Globally Live Variables

with `z` if `x` becomes defined (*i.e.*, a new instance of `x` is created) while `z` is globally live. In particular, if variable `x` becomes defined while `x` itself is globally live, then it interferes with itself.

7.4.1 Interference pairs

Interference analysis of a program is done by collecting information about which variables are live while other variables become defined. One piece of such information is called an *interference pair*, and may be *unconditional* of form (`z`,`x`) or *conditional* of form ($\mathtt{z},\mathtt{y} \gg \mathtt{x}$). The first form represents that `x` becomes defined while `z` is live, and is called *unconditional interference*. The second form is a special case of the first one where `x` becomes defined by a *copying* from variable `y` while `z` is live; this is called *conditional interference*. This terminology is from Sestoft [58].

Consider the application ($\mathtt{t}_1\ \mathtt{t}_2$). If $\mathtt{t}_2$ is bound to the value of some parameter `y` by `let t`$_2$ `= y in ...` and if `x` belongs to $\{\ \mathtt{x}^i_{j+1} \mid (\mathtt{f}^i,\ j) \in \mathrm{ca}[\![\mathtt{t}_1]\!]\ \}$ we have conditional interference and write (z,y$\gg$x) if z is live after the application.

A conditional interference pair (z,y$\gg$x) is so called because it is harmless on the condition that y and z, or y and x, are allocated as the same global variable.

To see this, first assume that variables y and z are allocated together. The interference pair (z,y$\gg$x) says that x becomes defined by copying from y while z is live. But this does not matter because y and z must have the same value by virtue of being allocated together, and so the value of z would not be changed by overwriting z along with x. Thus the conditional interference pair is harmless in this case.

Now assume that variables y and x are allocated together. The interference pair (z,y$\gg$x) says that x becomes defined by copying from y while z is live. But since x and z would have the same value if x and z were allocated as the same global variable, z and x and y would all have the same value, and updating x from y could not possibly change the value of z. Thus the conditional interference pair is harmless in this case also.

The conditional interference pairs (z,z$\gg$x) and (z,x$\gg$x) are harmless no matter how variables are grouped.

Conditional interference pairs are particularly useful when one wants to allocate several function parameters as the same global variable. This is further discussed in Section 7.5 below.

7.4.2 Computing interference information

The *interference analysis function* $\mathcal{I}$ for computing the interference pairs in a given program is shown in Figure 4. This is a slightly simplified analysis function which presumes all cases of interference are unconditional. Thus it is conservative in that it assumes that no cases of interference can be harmless.

To make analysis $\mathcal{I}$ take conditional interference into account, it must collect a conditional interference pair when the $\mathtt{t}_2$ in the rule for application is bound to a variable, and an unconditional interference pair when it is not. However, the mechanics of this is not shown in the figure.

The most interesting rule is that for application ($\mathtt{t}_1$ $\mathtt{t}_2$). Here an (unconditional) interference pair (z,x) is collected for every variable x of which a new instance may be created at the application, and every variable z which is live when the new instance is created.

The set of parameters for which a new instance can be created is approximated by the set "upds" of variables $\mathtt{x}^i_{m+1}$ for which $\mathtt{f}^i[v_1,\ldots,v_m]$ is a possible value of $\mathtt{t}_1$. The set of possible values of $\mathtt{t}_1$ is approximated using the closure analysis $\mathrm{ca}[\![\mathtt{t}_1]\!]$.

The set of parameters live at the creation is approximated by the set "live", which is the capture closure of the union of those live after the application (δ) and the variables $\mathtt{t}_1$ and $\mathtt{t}_2$.

$\mathtt{e}$: *Expr*	Sequentialized expressions (Section 7.2.3)
δ : $\mathit{VarSet} = \wp(\mathrm{Var})$	Set of live variables
θ : $\mathit{GLEnv} = \mathit{Fun} \rightarrow \wp(\mathit{Param})$	The avoid-set map
ca : $\mathit{ClMap} = \mathit{Expr} \rightarrow \wp(\mathit{AClo})$	The closure analysis (Section 7.2.5)
ccl: $\mathit{VarSet} \rightarrow \wp(\mathit{AClo})$	Capture closure (Section 7.3.3)
ξ : $\mathit{Inter} = \{\ (\mathtt{z},\mathtt{x}) \mid \mathtt{x}, \mathtt{z} \in \mathit{Param}\ \}$	Interference pairs

$$\mathcal{L} : \mathit{Expr} \rightarrow \mathit{VarSet} \rightarrow (\mathit{VarSet} \times \mathit{GLEnv})$$
$$\mathcal{I} : \mathit{Expr} \rightarrow \mathit{VarSet} \rightarrow \wp(\mathit{Inter})$$

$$\begin{array}{ll}
\mathcal{I}[\![\, \mathtt{x}^i_j \,]\!]\ \delta & = \{\} \\
\mathcal{I}[\![\, \mathtt{f}^i \,]\!]\ \delta & = \{\} \\
\mathcal{I}[\![\, \mathtt{if\ t\ e_1\ e_2} \,]\!]\ \delta & = (\mathcal{I}[\![\, \mathtt{e}_1 \,]\!]\ \delta) \cup (\mathcal{I}[\![\, \mathtt{e}_2 \,]\!]\ \delta) \\
\mathcal{I}[\![\, \mathtt{A(t_1,\ldots,t_n)} \,]\!]\ \delta & = \{\} \\
\mathcal{I}[\![\, \mathtt{let\ t_1\ =\ e_1\ in\ e} \,]\!]\ \delta & = (\mathcal{I}[\![\, \mathtt{e}_1 \,]\!]\ (\delta_1 \setminus \{\mathtt{t}_1\})) \cup (\mathcal{I}[\![\, \mathtt{e} \,]\!]\ \delta) \\
& \quad \text{where } (\delta_1,\theta_1) = \mathcal{L}[\![\, \mathtt{e} \,]\!]\ \delta) \\
\mathcal{I}[\![\, \mathtt{t_1\ t_2} \,]\!]\ \delta & = \{\ (\mathtt{z},\mathtt{x}) \mid \mathtt{z} \in \text{live}, \mathtt{x} \in \text{upds}\ \} \\
& \quad \text{where upds} = \{\ \mathtt{x}^i_{j+1} \mid (\mathtt{f}^i, j) \in \text{ca}[\![\, \mathtt{t}_1 \,]\!]\ \} \\
& \qquad\qquad \text{live} \;\; = \mathit{ccl}(\delta \cup \{\mathtt{t}_1, \mathtt{t}_2\})
\end{array}$$

The analysis result we seek is

$$\xi = \bigcup \{\ \mathcal{I}[\![\, \mathtt{e}^i \,]\!]\ (\theta\ \mathtt{f}^i) \mid i = 1, \ldots, n\ \}$$

where θ is computed by the analysis in Figure 3 and n is the total number of defined functions.

Figure 4: Unconditional Interference

7.5 Variable groups

A *variable group* γ is a set of variables that can be allocated as one global variable, that is, such that there is no interference between any two variables `z` and `x` in γ. A *variable grouping* Γ is a set of disjoint variable groups.

A set of variables can be grouped together if no two variables in the set interfere. Thus conditional and unconditional interference shapes the possible variable groups.

Proposition Let γ be a set of variables such that

1. there is no interference pair (z,x) with $\mathtt{z}, \mathtt{x} \in \gamma$.

2. for every conditional interference pair $(\mathtt{z},\mathtt{y}\gg\mathtt{x})$ with $\mathtt{z}, \mathtt{x} \in \gamma$, we have $\mathtt{y} \in \gamma$ also.

Then all the variables in γ can be globalized as one and the same global variable.

Proof: At every point where a variable $\mathtt{z} \in \gamma$ is live and another variable $\mathtt{x} \in \gamma$ becomes defined (by an assignment which could potentially disturb the value of z), really x is defined by a copy from $\mathtt{y} \in \gamma$ which must have the same value as $\mathtt{z} \in \gamma$. So z is not disturbed. □

If x is a globalizable variable, then the singleton {x} is a possible variable group. However, the larger the variable groups, the better. Unfortunately, it is easy to find a program where possible variable groupings are $\Gamma_1 = \{\{\mathtt{x},\mathtt{y}\},\{\mathtt{z}\}\}$ or $\Gamma_2 = \{\{\mathtt{x}\},\{\mathtt{y},\mathtt{z}\}\}$ but where $\Gamma = \{\{\mathtt{x},\mathtt{y},\mathtt{z}\}\}$ is not possible. Thus the set of possible variable groupings for a given program does not form a lattice under the ordering "inclusion of variable groups". This means that for a given program there is not necessarily a "best" variable grouping: there may be several distinct, maximal ones.

We use a simple heuristic: given two possible groupings Γ_1 and Γ_2, the better is that one which has most copying $\mathtt{y}\gg\mathtt{x}$ between variables belonging to the same group γ. The justification is that the copying will disappear when all variables of γ are replaced by one (global) variable. This is a heuristic because it does not take the actual run-time frequencies of the copyings into account, only the number of occurrences in the program.

An algorithm to choose a good grouping: Consider the set of globalizable variables $\{\mathtt{x}_n^m,\ldots,\mathtt{x}_j^i\}$. Construct the trivial grouping $\Gamma_0 = \{\{\mathtt{x}_n^m\},\ldots,\{\mathtt{x}_j^i\}\}$. While possible, let $\Gamma_{k+1} = (\Gamma_k \backslash \{\gamma_1,\gamma_2\}) \cup \{\gamma_1 \cup \gamma_2\}$ where γ_1 and γ_2 are chosen such that

1. $(\Gamma_k \backslash \{\gamma_1,\gamma_2\}) \cup \{\gamma_1 \cup \gamma_2\}$ is a variable grouping.
2. There does not exist two other groups γ_i, $\gamma_j \in \Gamma_k$ satisfying property 1. with more copying between variables in the two groups that between γ_1 and γ_2.

This algorithm works well in practice and is, *e.g.*, able to group all the store variables in a small interpreter.

7.6 Globalization

The interference analysis information is used to find a variable grouping Γ, which in turn is used in the globalization transformation. With globalization, the program is transformed to replace all uses of a variable x in a globalizable group γ by uses of one new global variable X_γ, and every creation of a new instance of x is replaced by an assignment to the new global X_γ.

To formalize this, the SECD machine is extended with a global store G and a new instruction (Glo X_γ) to access the global store, and the function application instruction App is modified to update the global store, as described in Section 7.6.3 below.

7.6.1 Transforming uses of parameters

Assume that variable x has been found to be globalizable and belongs to variable group $\gamma \in \Gamma$. Globalization means that instead of loading the value of x from the environment E, we will load the value of X_γ from the global store G. This is effected in the SECD machine code by replacing all instructions (Var x) by (Glo X_γ).

`Glo` X_γ; C	S	E	D	G	$\Rightarrow$	C	(G X_γ):S	E	D	G	
`App`; C	v_{m+1}:$\mathtt{f}^i[v_1,\ldots,v_m]$:S	E	D	G	$\Rightarrow$	C	$\mathtt{f}^i[v_1,\ldots,v_{m+1}]$:S	E	D	G′	if $m+1 < arity_i$
`App`; C	v_{m+1}:$\mathtt{f}^i[v_1,\ldots,v_m]$:S	E	D	G	$\Rightarrow$	C_i	[]	$[\mathtt{x}^i_j \mapsto v^i_j]$	(C,S,E):D	G′	if $m+1 = arity_i$

where G′= G$[X_\gamma \mapsto v_{m+1}]$ if $\exists\gamma \in \Gamma.\ \mathtt{x}^i_{m+1} \in \gamma$
and G′= G otherwise

Here C_i is the code for the body of defined function $\mathtt{f}^i$; and [] is the empty stack.

Figure 5: Store Handling SECD Machine Rules

7.6.2 Transforming definitions of parameters

Assume that variable $\mathtt{x}^i_{m+1}$ is globalizable and that $\mathtt{x}^i_{m+1} \in \gamma \in \Gamma$. Usually, a new instance of $\mathtt{x}^i_{m+1}$ is created at an application ($\mathtt{e}_1$ $\mathtt{e}_2$) when $\mathtt{e}_1$ evaluates to a closure $\mathtt{f}^i[v_1,\ldots,v_m]$. This creation of a new instance must be replaced by an assignment to the global variable X_γ. Therefore the instruction `App` is modified as shown in Section 7.6.3 below, so that it does this update at an application, whether partial or full.

Note that this modification makes the `App` instruction dependent on the variable grouping Γ computed. Thus a small additional element of interpretation has been introduced. In addition to testing whether the closure on the stack top is now fully applied, the `App` instruction must now check whether the variable $\mathtt{x}^i_{m+1}$, of which an instance is being created, is globalizable. If so, it must find which global variable X_γ to update.

Alternatively, one may require that either all the functions to which $\mathtt{e}_1$ may evaluate have their first argument globalizable as the same global variable, or none of them has globalizable first argument. This would allow the use of a specialized application instruction to do the assignment, but would give less opportunity for globalization.

7.6.3 Modifying the SECD machine

The modified rules for handling globalized variables in the SECD machine are shown in Figure 5. There are three changes: First, a global store G : *Param* $\rightarrow$ *Val* has been added to the state of the machine. Secondly, a new instruction (`Glo` X_γ) is introduced to refer to globalized variables. Finally, the rules for `App` are changed to have an additional effect on the global store when the variable $\mathtt{x}^i_{m+1}$ (of which a new instance is created) is globalizable. In this version of the machine we let `App` extend the closure or environment with the value v_{m+1} as well. This is superfluous for the purposes of running the transformed program, but is helpful in proving the correctness of globalization.

7.7 Correctness of globalization

First we show the liveness analysis correct with respect to a more precise notion *live*(C,S,E,D) of liveness based on the state of the SECD machine (Section 7.7.2).

Second, we show that if `x` is not in *live*(C,S,E,D) at any time a new instance of `x` is created during evaluation, then globalization of `x` is correct (Section 7.7.3).

Third, we know that if `x` does not interfere with itself according to the interference analysis, then it is not live according to the liveness analysis at any time a new instance of `x` may be created during evaluation. Combining this with the two previous observations, we know that it is correct to globalize `x` if it does not interfere with itself according to the interference analysis.

Below we shall assume throughout that the closure analysis ca⟦ `e` ⟧ gives a safe approximation (superset) of the closures that `e` can actually evaluate to. This has been proven elsewhere [58].

7.7.1 Sequentialization does not change the machine code

We have seen previously that the code $\mathcal{C_S}$⟦ `se` ⟧ generated for the sequentialized version `se` = $\mathcal{S}$⟦ `e` ⟧ of source expression `e` is exactly the same as the code $\mathcal{C}$⟦ `e` ⟧ generated for `e`. So evidently the sequentialized expression has precisely the same operational behaviour as the source expression. The only difference is that in the sequentialized expression this behaviour (order of evaluation) is made explicit already at the expression level.

This extends to globalized sequentialized programs. The global store updates introduced in a globalized program are done by the `App` instructions (whether partial or full applications). Since the application operations in `se` are done in exactly the same order and in the same places during execution in `se` as in `e`, it suffices to study their effect in `se`; any conclusions drawn for `se` are automatically valid for `e`.

7.7.2 Correctness of the liveness analysis

First we prove that the global liveness analysis result is correct: We prove that at every step in every evaluation of the program (with the abstract machine), whatever variables are live in the machine state (locally in C; indirectly locally in E or S; or globally (in the dump D)) are also live according to the analysis.

The avoid set (θ $\mathtt{f}^i$) for a function $\mathtt{f}^i$ is an approximation to the set of indirectly live variables in the dump's components, for all steps of evaluation of the function's body code C_i in all possible evaluations of the program.

To prove that the liveness analysis $\mathcal{L}$ yields safe liveness information we annotate the sequentialized program with the analysis results and let the compilation function $\mathcal{C_S}$ carry the annotations to the code. We then show that all possible machine states (C,S,E,D) have their sets of live variables contained in the set attached to the first instruction in the C component.

Given a sequentialized program with goal expression $\mathtt{e}_0$:

$$
\begin{array}{lll}
\mathtt{letrec}\ \mathtt{f}^1\ \mathtt{x}^1_1 \ldots \mathtt{x}^1_{arity_1} & = & \mathtt{e}^1 \\
\qquad\quad \ldots & & \\
\qquad\quad \mathtt{f}^n\ \mathtt{x}^n_1 \ldots \mathtt{x}^n_{arity_n} & = & \mathtt{e}^n \\
\mathtt{in}\ \mathtt{e}_0 & &
\end{array}
$$

Input to the program is through the free variables of $\mathtt{e}_0$. The output of the program is the value of $\mathtt{e}_0$. We require that the free variables as well as $\mathtt{e}_0$ itself have first order values: as usual, the input and output cannot contain functions.

When a program is compiled, the code $C_i = \mathcal{C_S}[\![\,\mathtt{e}^i\,]\!]$;`Ret` is generated for each function $\mathtt{f}^i$. The SECD machine is then started in state (C,[],E,[]) where $C_0 = \mathcal{C_S}[\![\,\mathtt{e}_0\,]\!]$;`Ret` and the input to the program is provided by the initial environment E.

Assume that the avoid-set map θ is computed as in Figure 3 with the additional requirement that $\theta \sqsupseteq \mathrm{snd}(\mathcal{L}[\![\,\mathtt{e}_0\,]\!]\bot_{GLEnv})$ to take the goal expression's contribution to the avoid-set map into account. For each subexpression `e` of each function body $\mathtt{e}_i$, compute $\Delta = \mathit{ccl}(\mathrm{fst}(\mathcal{L}[\![\,\mathtt{e}\,]\!](\theta\ \mathtt{f}^i)))$. For each subexpression `e` of the goal expression, compute $\Delta = \mathit{ccl}(\mathrm{fst}(\mathcal{L}[\![\,\mathtt{e}\,]\!]\{\}))$. Attach all these sets Δ to the respective expressions `e` for which they were found. Each expression in the program is thus annotated with a set which contains — according to the analysis – the variables that are live *before* evaluation of that expression. Now let the compilation function $\mathcal{C_S}$ place the Δ's as labels in front the generated instructions, *e.g.*:

$$\mathcal{C_S}[\![\,{}^{\Delta}(\mathtt{A}\ \mathtt{t}_1\ \ldots\ \mathtt{t}_n)\,]\!] = \Delta\text{: }\mathtt{Bas\ A}$$

Finally, label the last instruction "$(\theta\ \mathtt{f}^i)$: `Ret`" of each C_i (the code for $\mathtt{f}^i$) and "{}: `Ret`" of C_0 (the code for the goal expression). We now claim that the label at each instruction yields a safe approximation to the live variables in the current machine state during *any* computation.

What are the live variables, *live*(C,S,E,D), in a state (C,S,E,D)? The C component contributes the locally live variables, written *ll*(C), those that are accessed in C by the instruction `Var`. From the stack $S = v_m\!:\!..\!:\!v_1$ get the set of stack captured variables *sc*(S) $= \{\mathit{cap}(v_i) \mid i \in [1..m]\}$. The values of the locally live variables may also capture variables: $\mathit{capll}(C,E) = \bigcup \{\mathit{cap}(E(\mathtt{x})) \mid x \in \mathit{ll}(C)\}$. The dump D is either empty, D = [], or of form $D = (C',S',E')\!:\!D'$, and the set of live variables on the dump is defined accordingly:

$$\begin{array}{lcl}\mathit{dump\text{-}live}([\,]) & = & \{\} \\ \mathit{dump\text{-}live}((C',S',E')\!:\!D') & = & \mathit{live}(C',S',E',D')\end{array}$$

and *live*(C,S,E,D) is defined by

$$\begin{array}{lcl}\mathit{live}(C,S,E,D) & = & \mathit{ll}(C) \cup \mathit{sc}(S) \cup \mathit{capll}(C,E) \cup \\ & & \mathit{dump\text{-}live}(D)\end{array}$$

Proposition For all states $({}^{\Delta}C,S,E,D)$ that may arise during a computation, $\Delta \supseteq$ *live*(C,S,E,D).

Proof: We split the proof into two parts. First we will prove:

Claim 1. For every machine state $({}^{\Delta}C,S,E,D)$ where ${}^{\Delta}C$ is a part of the code for $\mathtt{f}^i$, if $(\theta\ \mathtt{f}^i) \supseteq \mathit{dump\text{-}live}(D)$ then $\Delta \supseteq \mathit{live}(C,S,E,D)$.

The first component of *live*(C,S,E,D), *ll*(C) is clearly contained in Δ since the occurrences of (`Var x`) in C correspond exactly to the occurrences of `x` in the sequentialized expression. Since the analysis result includes the *capture closure* of the variables, *capll*(C,E) is thus also included. In a sequentialized expression, the values that the machine push on the stack get bound to temporary variables `t` which are plain locally live until they are used (which corresponds to popping them from the stack in machine terminology). Hence $\Delta \supseteq sc(\mathrm{S})$. Last, the *dump-live* variables were assumed to be included in the avoid-set for $\mathtt{f}^i$ and thus in Δ.

> *Claim 2.* For every machine state $({}^{\Delta}\mathrm{C},\mathrm{S},\mathrm{E},\mathrm{D})$ where ${}^{\Delta}\mathrm{C}$ is a part of the code for $\mathtt{f}^i$, it holds that $(\theta\ \mathtt{f}^i) \supseteq$ *dump-live*(D).

We will prove this by induction on the depth of the dump D, *i.e.*, the depth of function calls during execution.

Base case: When D = [] the claim is trivially true.

Inductive case: Consider any state of form $(\mathrm{C}_i,[],[\mathtt{x}^i_j \mapsto v^i_j],(\mathrm{C},\mathrm{S},\mathrm{E}):\mathrm{D})$ where C is a part of the code C_k for function $\mathtt{f}^k$. Thus an application of function $\mathtt{f}^i$ is in progress, and its has been called from $\mathtt{f}^k$. The preceding state must have been $(\mathtt{App};\ \Delta{:}\mathrm{C},\ v_{m+1}{:}\mathtt{f}^i[v_1,\ldots,v_m]{:}\mathrm{S},\ \mathrm{E},\ \mathrm{D})$, and the state after return from function $\mathtt{f}^i$ will be $(\Delta{:}\mathrm{C},\ v{:}\mathrm{S},\ \mathrm{E},\ \mathrm{D})$, where v is the result of the application of $\mathtt{f}^i$. By the induction hypothesis we have $(\theta\ f^k) \supseteq$ *dump-live*(D) and hence by Claim 1: $\Delta \supseteq live(\Delta{:}\mathrm{C},v{:}\mathrm{S},\mathrm{E},\mathrm{D}) \supseteq live(\Delta{:}\mathrm{C},\mathrm{S},\mathrm{E},\mathrm{D})$.

Now we can prove the conclusion of the inductive case, that $(\theta\ \mathtt{f}^i) \supseteq$ *dump-live*((C,S,E):D), as follows:

$$\begin{aligned}
(\theta\ \mathtt{f}^i) &= (\textstyle\bigsqcup\{\mathrm{snd}(\mathcal{L}[\![\mathtt{e}^j]\!](\theta\ f^j)) \mid f^j \text{ is a function}\})\mathtt{f}^i \\
&\supseteq (\mathrm{snd}(\mathcal{L}[\![\mathtt{e}^k]\!]\ (\theta\ f^k)))\ \mathtt{f}^i \\
&\supseteq (\mathrm{snd}(\mathcal{L}[\![\mathtt{t}_1\ \mathtt{t}_2]\!]\ \Delta))\ \mathtt{f}^i \\
&\supseteq \Delta \\
&\supseteq live(\mathrm{C},\mathrm{S},\mathrm{E},\mathrm{D}) \\
&= dump\text{-}live((\mathrm{C},\mathrm{S},\mathrm{E}){:}\mathrm{D})
\end{aligned}$$

Here we exploit that θ is defined as a fixed point (line 1); the definition of $\mathcal{L}$ on applications and the fact that $(\mathtt{f}^i, arity_i\text{-}1) \in \mathrm{ca}[\![\mathtt{t}_1]\!]$ (line 4); the validity of Claim 1 as noted above (line 5); and the definition of *dump-live* (line 6).

We have now proved the inductive step, and hence Claim 2, which concludes the proof that the liveness analysis safely approximates the liveness of variables in the machine as defined by the function *live*.

7.7.3 Correctness of globalization

Next we shall prove globalization correct, exploiting the notion *live* for machine states (C,S,E,D) as defined in the previous section. Since the liveness analysis safely approximates *live*, we can conclude that the globalization decisions can safely be made on the basis of information from the liveness analysis.

Globalization is done by an extension of the `App` instruction as described in Section 7.6 and formalized in Figure 5. For the purposes of this correctness proof, we shall consider "semi-globalized" programs. In such programs, the global store G is kept updated by the extended `App` instruction, but the values in the global store are not yet used. Evidently this conservative extension cannot affect evaluations or their results, and hence this allows us to prove that the global store G accurately keeps track of the values of globalizable variables.

We assume that `x` belongs to a variable group γ such that for every two variables `x` and $\mathtt{x}^i_{m+1}$ in γ, and at every evaluation step (`App`; C,S,E,D) at which an instance of $\mathtt{x}^i_{m+1}$ is created, it holds that `x` is not in *live*(`App`; C,S,E,D),

We want to prove by contradiction that at any use (`Var x`) of `x` in an evaluation, (E `x`) = (G X_γ). Thus assume this is not the case: there is a use of `x` at which (E `x`) $\neq$ (G X_γ).

The "correct" value v of `x` in the environment E must come from some application `App`, which also updated G to v for X_γ. Since (G X_γ) is different from v at the use of `x`, there must have been some intervening redefinition of (G X_γ). This redefinition can be caused only by the evaluation of an application in machine state (`App`; C, v_{m+1}:$\mathtt{f}^i[v_1,...,v_m]$:S, E, D) for some (other) variable $\mathtt{x}^i_{m+1} \in \gamma$. But according to the Lemma below, at that time `x` must have belonged to *live*(C,S,E,D), which contradicts the assumption made about γ.

Thus it must hold at all uses (`Var x`) of $\mathtt{x} \in \gamma$ that (E `x`) = (G X_γ).

Lemma 2 At all machine states (C,S,E,D) from the application (`App`; C′, v:$\mathtt{f}^i[v_1 ... v_m]$:S′, E′, D′) which creates a new instance of variable $\mathtt{x}^i_{m+1}$, up until the last use of that instance, $\mathtt{x}^i_{m+1}$ belongs to *live*(C,S,E,D).

Proof outline: This is clearly the case just after the application. If the application is full, the next machine state will be (C_i,[],E″,D″), and either $\mathtt{x}^i_{m+1}$ occurs in C_i and thus will be in *ll*(C_i); or it does not, in which case it cannot have any later use. If there is a call from C_i to some (other) function, with a use of $\mathtt{x}^i_{m+1}$ in C_i after that call, then $\mathtt{x}^i_{m+1}$ will be in *dump-live* during the evaluation of the called function. The instance of $\mathtt{x}^i_{m+1}$ cannot have a use after the called function $\mathtt{f}^i$ returns.

Otherwise, if the application is partial, then in the next machine state the newly created instance of $\mathtt{x}^i_{m+1}$ will be in a closure on the stack (top), and thus $\mathtt{x}^i_{m+1}$ belongs to *sc*(S).

Let us examine what can happen to the closure containing the new instance of $\mathtt{x}^i_{m+1}$.

If the stack S is pushed onto the dump D, then as above, $\mathtt{x}^i_{m+1}$ belongs to *dump-live*((C,S,E):D) until the stack is popped off again.

If the closure is further applied, $\mathtt{x}^i_{m+1}$ will continue to belong to *sc*(S) until it is fully applied, in which case the situation described above obtains.

If the closure is moved to the environment E (as the value of a parameter w of some (other) function), then either there is a use of w in the current code C, in which case $\mathtt{x}^i_{m+1}$ belongs to *capll*(C,E), or there is no later use of w, in which case there cannot be a later use of $\mathtt{x}^i_{m+1}$.

From the environment E, a copy of the closure may be pushed onto the stack S by a use of variable w. This will make $\mathtt{x}^i_{m+1}$ belong to *sc*(S). □

7.8 Experiments

In this section we show the avoid sets and variable groupings computed for three example programs by our Miranda™ implementation of the analyses.

Globalizing a captured variable

Source program:

```
map f l = if   (null? l)
          then ()
          else (cons (f (car l)) (map f (cdr l)))
add n x = (plus n x)
goal  i = (append (map (add 2) i) (map (add 3) i))
```

The avoid sets, *i.e.*, the θ map:

$$\begin{aligned}
\theta(\texttt{add}) &= \{\texttt{f},\texttt{i},\texttt{l},\texttt{n}\}\\
\theta(\texttt{goal}) &= \{\}\\
\theta(\texttt{map}) &= \{\texttt{i}\}
\end{aligned}$$

The variable grouping:

$$\{\{\texttt{f}\},\{\texttt{i}\},\{\texttt{l}\},\{\texttt{n}\},\{\texttt{x}\}\}$$

A small interpreter for an imperative language

The interpreter below interprets the language

$$\begin{aligned}
c &::= \text{while } \texttt{e} \text{ do } c \mid v := \texttt{e} \mid c_1;c_2\\
\texttt{e} &::= \texttt{e}_1 + \texttt{e}_2 \mid v
\end{aligned}$$

Source program:

```
cmd c = if (while? c)
        then (dowhile (take-cond c) (take-cmd c))
        else if (assignment? c)
             then (upd (take-id c) (take-exp c))
             else if (sequence? c)
                  then (seq (fst-cmd c) (snd-cmd c))
                  else nop
dowhile ed cd sd = if (zero? (exp ed sd))
                   then sd
                   else (dowhile ed cd (cmd cd sd))
upd idu eu su    = (update idu (exp eu su) su)
seq c1 c2 sc     = (cmd c2 (cmd c1 sc))
exp e            = if (number? e)
                   then (add (fst-exp e) (snd-exp e))
                   else (look e)
look idl sl      = (access idl sl)
add e1 e2 sa     = (plus (exp e1 sa) (exp e2 sa))
nop sn           = sn
main cm          = (cmd cm initial-store)
```

The avoid sets:

$$
\begin{array}{lcl}
\theta(\texttt{cmd}) & = & \{\texttt{c1,c2,cd,ed,eu,idu,sc,sd}\}\\
\theta(\texttt{dowhile}) & = & \{\texttt{c1,c2,cd,ed,eu,idu}\}\\
\theta(\texttt{upd}) & = & \{\texttt{c1,c2,cd,ed,eu,idu}\}\\
\theta(\texttt{seq}) & = & \{\texttt{c1,c2,cd,ed,eu,idu}\}\\
\theta(\texttt{exp}) & = & \{\texttt{c1,c2,cd,e2,ed,eu,idu,sa,sd,su}\}\\
\theta(\texttt{look}) & = & \{\texttt{c1,c2,cd,e2,ed,eu,idu,sa,sd,su}\}\\
\theta(\texttt{add}) & = & \{\texttt{c1,c2,cd,e2,ed,eu,idu,sa,sd,su}\}\\
\theta(\texttt{nop}) & = & \{\texttt{c1,c2,cd,ed,eu,idu}\}\\
\theta(\texttt{main}) & = & \{\}
\end{array}
$$

The variable grouping:

$$\{\{\texttt{c,cm,e,idl}\},\{\texttt{e1}\},\{\texttt{sa,sc,sd,sl,sn,su}\}\}$$

Note that all the store parameters are replaced by one global variable. Note also that (surprisingly!) many other parameters may be globalized. This is a tendency in many of the experiments we have made: parameters other than "those we are after" turn out to be globalizable.

Composing a list of functions

The function `compose` composes the functions contained in the list given as arguments. Source program:

```
compose fs c = if   (null? fs)
               then c
               else ((car fs) (compose (cdr fs) c))
twice   f  x = (f (f x))
double  y    = (plus y y)
main    m    = (compose (cons (twice double)
                              (cons double ())) m)
```

The avoid sets:

$$
\begin{array}{lcl}
\theta(\texttt{compose}) & = & \{\texttt{f}\}\\
\theta(\texttt{twice}) & = & \{\texttt{f}\}\\
\theta(\texttt{double}) & = & \{\texttt{f}\}\\
\theta(\texttt{main}) & = & \{\}
\end{array}
$$

The variable grouping:

$$\{\{\texttt{c,m,x,y}\},\{\texttt{f}\},\{\texttt{fs}\}\}$$

In this program, all function parameters may thus be replaced by three global variables.

7.9 Related work

The work reported here continues Schmidt's work on detecting single-threaded store variables in denotational definitions [57]. However, Schmidt expressed the single-threading criteria as simple and elegant conditions on the types assigned to expressions in a simply typed lambda calculus. His criteria do not allow globalizable variables to be captured in closures (*i.e.*, in free variables under lambdas). Our attempts to lift this restriction while retaining the simpler conditions failed, but were the starting point of this work. The present work also extends Schmidt's by giving an algorithm to construct variable groupings, and our methods do not require the language to have a type structure.

Also closely related is Sestoft's previous work on replacing function parameters by global variables, which was based on analysis of the possible definition-use paths for a program. That method required a fixed evaluation order and was able to exploit the extra information that gives. Sestoft's previous work had the same restriction as Schmidt's: globalizable variables were not allowed to be captured in closures. Extending the method of definition-use paths to allow this turned out to be too complicated, which is why we turned to the concept of (globally) live variable as used in this work.

Fradet's single-threading analysis is based on a continuation-passing form of typed expressions, which gives rather simple conditions on syntax and types. In this respect his work is quite similar to Schmidt's, but requires fixed evaluation order and is able to exploit it. Unlike the present work, but like all previous work, Fradet's method does not allow globalizable variables to be captured in closures.

Moreover, while to our knowledge the mentioned previous works have remained unimplemented, we have constructed a working prototype implementation of the present analysis algorithms in Miranda$^{\mathrm{TM}}$. However, the implementation is not documented in this report.

Chapter 8

Conclusion

We have developed several analyses and transformations for pure functional languages. The analyses are used to support program transformations, or to support other analyses which in turn support transformations. The purpose of the program transformations is to improve the programs by reducing their run time or storage consumption.

The analyses collect operational information: information about the possible run time behaviours of the analysed program. Some of the analyses (closure analysis, usage count analysis, interference analysis) have been implemented. The closure analysis, the interference analysis, and the globalization optimization have been proved correct. The effect of optimizations based on the usage interval analysis have been tested on an experimental implementation of a functional language.

Below we explain our contributions and conclusions, and outline future work for each chapter of the report. A quick summary of the work can be obtained by reading only the summary sections.

8.1 A lazy example language

8.1.1 Summary

We have designed a working implementation of an untyped lazy higher order functional language L with base values and lazy data types. The implementation is based on a variant of the simple Krivine abstract machine and the encoding of data structures as higher order functions. This somewhat theoretical implementation works well, and in particular lazy printing of data structures was interesting (and easy) to implement. The full details of the implementation are given in the appendices.

8.1.2 Future work

We have claimed that the K machine is a proper choice of abstract machine for execution of the lazy language L, and a suitable basis for proving intensional analyses of L correct. To support this claim, we should prove that the machine does implement outermost weak reduction.

8.2 Closure analysis

8.2.1 Summary

We have described and implemented a closure analysis (for the lazy language L) which computes for an expression `e` an approximation to the set of functions `e` can evaluate to. The analysis is proved correct with respect to the abstract machine for implementing L. In contrast to earlier versions of closure analysis, the present one handles `letrec` bound variables (nullary functions). It also handles data structures via their encoding as functions, albeit with some loss of precision. The implementation is included in full.

We show how to extend first order analyses to higher order analyses using the closure analysis, and compare that to the "strictness pairs" method of Hudak and Young. The closure analysis method has much lower complexity but is much less precise.

8.2.2 Future work

The precision of closure analysis, especially as applied to data structures, should be studied more carefully. This might show that it is worth-while to further develop closure analysis for (typed) data structures in their own right, not their encoding as functions.

It should also be investigated whether closure analysis is suitable for extending a version of evaluation order analysis (Chapter 6) to higher order languages.

8.3 Usage interval analysis

8.3.1 Summary

We have designed and implemented a simple usage count analysis (for the lazy language L) which finds an approximation to the number of times a variable gets used in one evaluation of the containing expression. The analysis essentially is first order, and is extended to higher order using the closure analysis just described. The analysis works for data structures via their encoding as functions, but may give very conservative results.

The information collected by usage interval analysis can be used for optimization of suspensions (or, thunks), and we have designed and implemented two optimizing transformations for the lazy language L. One transforms call by need into call by name (for unshared arguments), the other transforms call by need into call by value (for strict arguments). Measurements on the improved implementation of L shows that a considerable part of the bookkeeping necessary for call by need can be avoided by program analysis. However, in the simple experimental implementation, the run time consumption is not much reduced by these optimizations. The implementations of the analysis and transformations are included in full.

8.3.2 Future work

The abstract machine used for implementing L cannot very well exploit strictness information. Hence a better study of optimizations based on usage interval analysis would require a more realistic machine, such as the Spineless Tagless G-machine by Peyton Jones [52].

The relation between usage interval analysis and the "update analysis" needed (but so far missing) for Peyton Jones's STG language should be clarified [52]. Possibly an update analysis could be developed from much the same ideas as the usage interval analysis.

A far more precise (and expensive) usage interval analysis, restricted to first order simply typed languages, could be achieved using the evaluation order analysis framework of Chapter 6. For this purpose, we let an evaluation order description in $\mathit{Eod}(\mathtt{e})$ for expression $\mathtt{e}$ be a *usage interval map*, mapping each variable $\mathit{Vars}(\mathtt{e})$ of $\mathtt{e}$ to its usage interval. Likewise, an argument eod in $\mathit{AEod}(n)$ is a mapping from the set $\{1, \ldots, n\}$ to the usage interval of the i'th argument. The ordering on these lattices is pointwise usage interval inclusion.

The plumbing functions for the analysis function $\mathcal{R}$ are also quite straightforward. For instance, $R_{var}(\mathtt{x})$ maps $\mathtt{x}$ to $[\mathit{One},\mathit{One}]$ and all other variables to $[\mathit{Zero},\mathit{Zero}]$, and R_{An} adds all the argument usage interval maps pointwise (*i.e.*, for each variable), and so on. Combination of usage interval maps, and extraction of argument eod's, and the entire framework of uniform evaluation order types *etc.* seem to carry over to this application also.

What is more, like the other analyses in the framework, this kind of usage interval analysis would be justifiable with respect to the occurrence path semantics. All the details of this remain to be investigated.

8.4 Evaluation order analysis

8.4.1 Summary

We have developed a backwards path analysis for a simply typed first order lazy language with data structures. A path analysis is an evaluation order analysis: it provides information about subexpression evaluation order in a lazy language. Our backwards path analysis gives such information even for expressions involving lazy data structures, which previous forwards path analyses do not. The analysis is not implemented or proven correct.

The backwards path analysis fits into an entire framework that accommodates a more fine-grained occurrence path analysis, a less precise but faster evaluation order analysis based on so-called evaluation order relations, a strictness analysis for data structures, and the improved usage interval analysis briefly suggested above. The framework is parametrized on the lattices of evaluation order descriptions and certain combining operations of the analysis.

The framework also provides a general way to obtain a finite lattice of contexts for expressions of recursive data types.

8.4.2 Future work

The framework could be extended in two directions: application to higher order languages, and polymorphic evaluation order types. (Recall that the example language is first order and simply typed).

The analysis of higher order functions could possibly be achieved using a closure analysis (see Chapter 4) or Hughes's general method for extending backwards analyses to higher order languages [38].

The extension to polymorphic evaluation order types would allow more general and more efficient analysis of the typically polymorphic functions used in functional programming. One example is `append` of lists of lists, where the evaluation order of the inner lists is immaterial to the analysis of `append`. Consider its polymorphic type, `append`: `(List` α`)` $\times$ `(List` α`)` $\rightarrow$ `(List` α`)`. Since `append` does not inspect the list elements, these may be of any type α. Moreover, for the same reason `append` cannot change the order of evaluation of components of values of type α; hence any evaluation order relation on these will be preserved by `append`. In fact, we conjecture that functions which are polymorphic in some component of their types are also evaluation order polymorphic in that component, much as conjectured for Hughes's backwards analysis framework.

8.4.3 Cheaper approximations?

Variable path analysis is neat but has far too high complexity for practical use, so useful approximations should be sought. The analysis based on evaluation order relations is a good candidate. However, its formal relation to the occurrence path analysis should be investigated. For example, the various operations (combination, the plumbing functions, the argument eod extraction) on evaluation order relations might be proved correct with respect to the corresponding operations on occurrence paths, via the concretization and abstraction functions. This would provide a proof that the analysis results are correct with respect to those of occurrence path analysis.

We should consider also which practical uses of evaluation order information there are, and whether the analysis is sufficiently precise for these uses. An implementation, possibly of a sticky version of the analysis, would shed light on its practical feasibility.

8.4.4 Semantic basis?

The occurrence path analysis should be justified with respect to a simple abstract machine such as the Krivine machine (see Section 3.2 or [16]). A computation with that machine gives a (finite or infinite) sequence of states, which fits well with the sequences of events obtained from the occurrence path analysis.

In the Krivine machine the event of expression `e` reaching its whnf is when the code part of the machine state reaches a `Lam` or `Ret` instruction, ready to take an argument off the same computation stack, which must be the same as when evaluation of `e` was initiated. Thus a relation between the analysis and the machine would build on the concepts of balanced trace and whnf computation from Section 4.2.

Clearly a standard denotational semantics would not do as a formal model, since it is devoid of all notions of time. Instead of using an abstract machine, one might use an instrumented semantics, where the meaning of an expression is a result value together with a path of the "actions" (such as variable accesses) needed to reach that result. This is the idea in Bloss and Hudak's path semantics [8], but the context-dependent evaluation order of lazy data structures would complicate the instrumented semantics considerably, and make it less obvious whether it is correct.

8.5 Globalization for partial applications

8.5.1 Summary

We have designed and implemented an interference analysis and a globalization transformation for an untyped strict higher order functional language H. The interference analysis finds which function parameters can be replaced by global variables by the subsequent globalization transformation. The analysis can recognize globalizable variables even if they become enclosed in partial applications (or equivalently, are free in lambda expressions). Previous analyses were unable to do this. We have proven the analysis and the globalization transformation correct with respect to a variant of Landin's SECD abstract machine, which implements the strict language H.

Our experiments show that a surprisingly high number of function parameters can be replaced by global variables, and also that our method actually extends the earlier ones in the handling of parameters that are captured in closures.

8.5.2 Future work

The central concept of interference is based on that of live variable, which in turn depends on the order of evaluation. Thus we assumed that our language is strict and has simple left-to-right evaluation order. By changing the translation to sequential expressions we can treat also languages with other fixed evaluation orders. To handle (strict) languages with indeterminate order of evaluation, such as Scheme, would require more modifications to the analysis, and the sequentialized expressions will be of little help.

In non-strict languages with data structures, subexpression evaluation order depends on the context, which makes analysis of evaluation order somewhat harder. Evaluation order analyses exist (see [6] and Chapter 6), and could presumably be used to determine the set of live variables at a given program point. Thus globalization might be extended to lazy languages, but the globalization of an expression would be valid only with respect to certain contexts of use.

Globalization of intermediate variables is another interesting extension, which would amount to sophisticated interprocedural register allocation. In our framework this extension could be achieved by globalizing the let-variables `t` of a sequentialized program.

Appendix A

Some Terminology

abstract machine An abstract machine is a state transition system.

backwards Backwards analysis *information* about an expression is information about what will (later) happen to (the result of) the expression. This is often called the context of the expression. A backwards analysis *method* is one which computes the contexts of the subexpression of an expression from the context of the expression.

base semantics The "essential" or un-instrumented semantics of a programming language, prescribing only its input-output behaviour. (Term from Søndergaard [65]).

call by name A (non-strict) call mechanism in which an argument expression is evaluated every time the corresponding formal parameter is used, *i.e.*, zero, one, or many times.

call by need A (non-strict) call mechanism in which an argument expression is evaluated only if the corresponding formal parameter is used, and only at its first use, *i.e.*, zero or one times.

call by value A (strict) call mechanism in which an argument expression is evaluated ahead of all uses of the corresponding formal parameter, and regardless whether it is actually used, *i.e.*, always once.

cell turnover is the number of allocation actions done during a run. Compare residency.

closure A closure `C`•E is a piece of code `C` plus bindings E for the free variables of `C`.

compile time (or compilation time) is the time (stage) at which compilation is done.

compile time analysis (or *static analysis*) is program analysis done without the program's run time input, that is, without running the program.

denotation The object(s) referred to by an expression.

extension Same as denotation, cf. intension.

extensional Having to do with the base semantics for a language (which describes only the input-output behaviour of a program). Analysis information is extensional if it is meaningful with reference exclusively to *what* results a computation may give and without reference to *how* these results were obtained. The correctness of an extensional analysis can be defined on basis of a program's input-output behaviour alone.

head normal form A core L expression is in head normal form if it has form $\lambda \mathtt{x}_1 \ldots . \lambda \mathtt{x}_n . (\mathtt{x}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_m)$ or $\lambda \mathtt{x}_1 \ldots . \lambda \mathtt{x}_n . (\mathtt{c}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_m)$ where `x` is a variable, $(\mathtt{c}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_m)$ is a partial application of a built-in function, and $m, n \geq 0$. (Adapted for core L from Peyton Jones [51]).

instrumented semantics A semantics giving information besides the input-output behaviour of a program. Usually, the extra information describes *how* the result of a computation is obtained, besides *what* it is.

intension The set of characteristics of the object(s) referred to by an expression, roughly *how* instead of *what*.

intensional Analysis information is intensional if it must be understood with reference to the *how* of a computation, not only the *what*: the results produced by a computation. Thus the correctness of an intensional analysis must be justified with respect to a semantics giving some information besides the input-output behaviour of a program.

lazy language A language in which functions and data structure constructors evaluate their arguments by need.

operational analysis information is the same as intensional analysis information.

residency The number of live cells (the heap consumption) at a given point in time [51]. Compare cell turnover.

run (noun) The execution or evaluation of a program.

run time The time (stage) at which a program is run.

run time consumption The amount of time it takes to run a program.

storage consumption see cell turnover and residency.

suspension or *thunk*. A closure `C`$\bullet$E constructed to suspend the evaluation of an expression; evaluation of the code `C` in environment E will produce the value of the suspended expression.

weak head normal form or *whnf*. A core L expression is in weak head normal form (whnf) if it has no top level redex, *i.e.*, if it is $\lambda \mathtt{x} . \mathtt{e}$, or an application of a variable $(\mathtt{x}\ \mathtt{e}_1\ \ldots\ \mathtt{e}_m)$ where $m \geq 0$. (Adapted for core L from Peyton Jones [51]).

Appendix B

Programs

The lazy language L discussed in Chapter 3 has been implemented for making experiments with closure analysis, usage count analysis, and the call by need to call by name optimization. This appendix briefly describes these implementations and lists the programs. Section B.1 describes L and the K machine, Section B.2 describes the closure analysis, Section B.3 describes the usage count analysis, and Section B.4 describes the improved code generation and a refined version of the K machine.

All programs concerning L and analysis of L have been written in Chez Scheme, which is an impure, untyped functional language in the Lisp tradition [19]. To make programs easier to write and read, we have used a package of syntactic extensions (function definition by cases, *etc.*) developed by Hans Dybkjær at DIKU. The pattern matching notation should be easily understandable.

Scheme was chosen for implementation language because it allows the various data structures with pointers to be encoded quite straightforwardly, it includes garbage collection, and it allows the destructive updates necessary for laziness. Moreover, it can be executed quite efficiently on our computers, which is necessary to make interpretation of non-trivial L programs feasible. Another implementation language with the same properties is Standard ML, which has the added advantage of modules, a type system, and handling pointers (references) in a more explicit and less error-prone manner [46]. Admittedly, the programs would have been "cleaner" and probably more readable if written in Standard ML. I should also offer an apology for using the arcane "backquote" notation: it *does* make program-generating programs more compact, and also more readable once you have got used to it.

B.1 Compilation of L and the K machine

The implementation of L consists of a compiler `l2core` from full L to core L, a compiler `core2k` from core L to K machine code, and an interpreter `k` for the K machine as summarized in Section 3.7.

The implementation is organized in three files: `compile.ss` contains the compiler from L to core L, file `kmachine.ss` contains the compiler from core L to K, and the K machine interpreter, and file `kaux.ss` contains auxiliary functions for the interpreter. Finally there is a file `auxiliary.ss` with general auxiliary functions.

B.1.1 How to run L programs

To compile and run an L program `e`, load file `loadk` and execute `(run e)` from the Scheme system prompt.

Example 13 A program to compute the squares of the integers from 20 down to 1:

```
> (load "loadk")
...
> (run '(letrec (downfrom n = (if (= n 0) Nil (Cons n (downfrom (- n 1)))))
                (square x = (* x x))
                (map f xs = (case xs of
                                  Nil         => Nil
                                  (Cons y ys) => (Cons (f y) (map f ys))))
                in (map square (downfrom 20))))
(400 361 324 289 256 225 196 169 144 121 100 81 64 49 36 25 16 9 4 1)
K machine terminated.
(time (k term ...))
    no collections
    130 ms elapsed cpu time
    633 ms elapsed real time
    11648 bytes allocated
```

□

B.1.2 How to make measurements

As can be seen from the example above, the following statistics are printed after every (terminating) computation: The cpu time in milliseconds (and the time spent in garbage collection if any), the wall clock time in milliseconds, and the cell turnover during the computation.

Furthermore, the K machine interpreter has a profiling mechanism to measure the frequency of execution of the various instructions (`App`, `Lam`, ...). Profiling is switched on by `(doprof)`, and off by `(noprof)`. When the interpreter is loaded it is switched off. When it is on, profiling information will be printed at the end of every (terminating) computation. For computations which are interrupted manually, such as the printing of an infinite list, the the profiling information can be requested by typing `(printprofile)` after the interrupt.

There is also a mechanism to report the residency, *i.e.* the amount of live cells in the heap, just after every garbage collection. This is turned on with the command `(doheap)` and off with `(noheap)` and is off when the interpreter is loaded. The residency in bytes is given brackets: `[1504]`.

Finally, the execution of K machine instructions can be traced. Tracing is turned on with the command `(dotrace)` and off with `(notrace)` and is off when the interpreter is loaded. Tracing produces a lot of output and is likely to make ordinary (especially data structure) output unreadable.

B.1.3 Program files

File `compile.ss` contains the compilation function from L to core L, as well as a description of the concrete (Scheme like) syntax of these languages.

```
; File "compile.ss" -- Peter Sestoft 1991-01-09, 1991-06-17

; Compilation of L programs into the L core language

; Representation of the full L source language
; <exp> ::= (lam <var> <exp>)            Abstraction
;         | <var>                        Variable or function
;         | (<exp> <exp> *)              Application
;         | <basefun>                    Such as +, -, *, /, not, =?, ...
;         | (quote <s-expression>)       Base constants
;         | <numeral>                    Numerals
;         | (if <exp> <exp> <exp>)       Conditional
;         | Nil | ()                     Empty list
;         | Cons | ::                    List constructor
;         | hd | tl                      Head and tail selectors
;         | (case <exp> of Nil => <exp> (Cons <var> <var>) => <exp>)
;         | (case <exp> of ()  => <exp> (::   <var> <var>) => <exp>)
;         | (letrec <def>* in <exp>)     Local recursive definitions
;         | (<exp> where <def>*)         Local recursive definitions
; <def> ::= <var> <var>* = <exp>         Variable or functions definition

; Representation of the L core language
; <cex> ::= (lam <var> <cex>)            Abstraction
;         | (var <sym>)                  Variable
;         | (app <cex> <cex>)            Application
;         | (bapp <baseop> <cex>*)       Base constants and functions
;         | (if <cex> <cex> <cex>)       Conditional
;         | (letrec <cdf>* <cex>)        Local recursive definitions
; <cdf> ::= (<var> = <cex>)              Variable or function definition

; COMPILATION FROM THE L SOURCE LANGUAGE INTO THE L CORE LANGUAGE
; -------------------------------------------------------------

(fun l2core (e) =
(match (e) by
     [('if e1 e2 e3)     => `(if ,(l2core e1) ,(l2core e2) ,(l2core e3))]
     [('hd e1)           => (l2core `(,e1 () (lam x (lam y x))))]
     [('tl e1)           => (l2core `(,e1 () (lam x (lam y y))))]
     ['Nil               => (l2core `(lam n (lam c n)))]
     ['()                => (l2core `Nil)]
     ['Cons              => (l2core `(lam x (lam y (lam n (lam c (c x y))))))]
     ['::                => (l2core `Cons)]
     ['hd                => (l2core `(lam z (hd z)))]
     ['tl                => (l2core `(lam z (tl z)))]
     [('quote c)         => `(bapp ,c)]
     [('lam v e1)        => `(lam ,v ,(l2core e1))]
     [('case e0 'of '()  '=> e1 ('::   v1 v2) '=> e2) &
                                           (and (atom? v1) (atom? v2))
                         => (l2core `(,e0 ,e1 (lam ,v1 (lam ,v2 ,e2))))]
     [('case e0 'of 'Nil '=> e1 ('Cons v1 v2) '=> e2) &
                                           (and (atom? v1) (atom? v2))
                         => (l2core `(,e0 ,e1 (lam ,v1 (lam ,v2 ,e2))))]
     [('letrec . dinb)   => (l2c-letrec dinb ())]
     [(e0 'where . defs) => (l2core ('letrec :: (append defs `(in ,e0))))]
     [v & (number? v)    => `(bapp ,v)]
     [v & (atom? v)      =>
        (match ((assoc v *basefun*)) by
               [(_ '1 . op) => (l2core `(lam x (,v x)))]
               [(_ '2 . op) => (l2core `(lam x (lam y (,v x y))))]
```

```
                [#f          => `(var ,v)])]            ; It must be a variable
     [(e0 . er)          =>
         (match ((assoc e0 *basefun*) er) by
              [(_ '1 . op) (e1) => `(bapp ,op ,(l2core e1))]
              [(_ '2 . op) (e1 e2) => `(bapp ,op ,(l2core e1) ,(l2core e2))]
              [ _ _ => (sfoldl mkapp (l2core e0) (map l2core er))])]
))

; Compiling applications

(fun mkapp (t1 t2) = `(app ,t1 ,t2))

; Compile local recursive definitions.

(fun l2c-letrec (dinb defs)
     [('in e0)          _ => `(letrec ,(reverse defs) ,(l2core e0))]
     [((xi . def) . r) _ =>
           (l2c-letrec r (`(,xi = ,(l2core (lambdify-def def))) :: defs))]
     [_ _           => (error 'l2c-letrec "Local definition syntax: ~s" dinb)]
)

; Convert given formal parameters in letrec definitions into lambda variables

(fun lambdify-def (def)
     [('= body)     => body]
     [('= . _)      => (error 'lambdify-def "Local definition syntax: ~s" def)]
     [(v . r) & (atom? v) => `(lam ,v ,(lambdify-def r))]
     [_             => (error 'lambdify-def "Local definition syntax: ~s" def)]
)

; DEFINING STRICT BASE FUNCTIONS
; -----------------------------

(fun mkbase1 (name op) = (name :: 1 :: op))
(fun mkbase2 (name op) = (name :: 2 :: op))
(define *basefun*
   (list (mkbase2 '+ +)
         (mkbase2 '- -)
         (mkbase2 '* *)
         (mkbase2 '/ /)
         (mkbase2 '= equal?)
         (mkbase2 '< <)
         (mkbase1 'not not)
         (mkbase2 'remainder remainder)
         (mkbase2 'quotient quotient)
))
```

B.1.4 The K machine

File `kmachine.ss` contains a description of K machine instructions, the compilation from core L to K instructions, and the K machine interpreter.

```
; File "kmachine.ss" -- Peter Sestoft, 1991-01-09, 1991-06-18
; Uses "auxiliary", "compile", "kaux"

; Interpretive implementation of the K machine (a lazy Krivine machine).
; This is the basic version without any optimized instructions.

; The K machine instructions are as follows:
; (These instructions are named as in Chapter 4 of the report).
```

```
; <ins> ::= (lam <ins>)                    Abstraction (deBruijn binding)
;         | (var <num>)                    Variable (deBruijn index)
;         | (app <ins> <ins>)              Application
;         | (cst <scheme-value>)           Base constants
;         | (b1 <op>) | (b2 <op>) | (r1) Base function evaluation
;         | (if)                           Conditional
;         | (letrec (<ins> *) <ins>)       Local recursive definitions
;         | (ret)                          Base value
;         | (pn)                           PN (lazy print instruction)
;         | (pnp)                          PN'
;         | (pc)                           PC
;         | (pcp)                          PC'
;         | (pcr)                          PCR
;         | (stop)                         Stopping the K machine

; MAIN FUNCTIONS TO COMPILE, RUN, AND TIME PROGRAMS
; ----------------------------------------------

(fun run (exp) =
    (run&time k (kcomp exp)))

(fun kcomp (e) = (core2k (l2core e) ()))

; COMPILATION FROM CORE L TO K INSTRUCTIONS

(fun core2k (e env) =
(match (e) by
     [('lam v e1)       => `(lam ,(core2k e1 (v :: env)))]
     [('var v)          => `(var ,(getindex v env))]
     [('app e1 e2)      => `(app ,(core2k e1 env) ,(core2k e2 env))]
     [('bapp op)        => `(cst ,op)]
     [('bapp op e1)     => `(app ,(core2k e1 env) (b1 ,op))]
     [('bapp op e1 e2)  => `(app (app (app ,(core2k e1 env) (r1))
                                      ,(core2k e2 env))
                                 (b2 ,op))]
     [('if e1 e2 e3)    => `(app (app (app ,(core2k e1 env) (if))
                                      ,(core2k e2 env))
                                 ,(core2k e3 env))]
     [('letrec defs e0) => (c2k-letrec defs e0 env ())]
))

; Variable lookup in the compile time environment

(fun getindex (v env) =
     (fun get-aux (env index)
          [()       _ => (error 'core2k "Variable ~s is undefined" v)]
          [(v1 . r) _ & (v1 =? v) => index]
          [(_  . r) _ => (get-aux r (+ 1 index))])
     (get-aux env 0)
)

; Compiling letrec definitions from core L to K

(fun c2k-letrec (defs e0 env rhss)
     [() _ _ _            => `(letrec
                               ,(map (lambda (e) (core2k e env)) (reverse rhss))
                                ,(core2k e0 env))]
     [((xi '= ei). r)_ _ _ => (c2k-letrec r e0 (xi :: env) (ei :: rhss))]
)

; REPRESENTATION OF DATA STRUCTURES IN THE K MACHINE:
; ------------------------------------------------
; WARNING: Heavy use is made of the fact that one can obtain pointers to
; data structures and their substructures, and that these can be modified
; destructively.  This is particularly important for the updating of
; markers, and for the creation of a recursive environment for letrec.
```

```
; An environment is a Scheme list (u0 u1 ... un) of closures
; <env> ::= <clo>*

; A closure is a Scheme pair of a K machine instruction and an environment
; <clo> ::= (<ins> . <env>)

; The K machine stack is represented as a vector s of closures or markers,
; together with a stack top pointer p.
; The stack is accessed using the macros sr and ss:  (sr 0) denotes the
; the stack top element, (sr 1) the element below the stack top, etc;
; (ss 0 val) sets the stack top to val, etc:
;     ... s[p+1]  s[p]   s[p-1]   ...     s[0]
;     ... (sr -1) (sr 0) (sr 1)   ...     bottom

; A marker is a pair of the tag "mrk" and a *pointer* to a closure.
; <mrk> ::= (mrk . <clo>)

; THE K MACHINE INTERPRETATION FUNCTION
; ------------------------------------

(fun k (t e s p) =
  (if *doprof* (prof! t))
  (if *dotrace* (print (car t) " "))
  (record-case t
   [app  (m n) (ss -1 (n :: e)) (k m e s (+ 1 p))]
   [var  (i)   (mrk&enter (list-ref e i) s p)]
   [lam  (m)   (let ((p (updatemrks! t e s p)))
                  (k m ((sr 0) :: e) s (- p 1)))]
   [r1   ()    (let ((u1 (sr 0)) (u2 (sr 1)) (uo (sr 2)))
                  (ss 2 u1) (ss 1 uo) (enter u2 s (- p 1)))]
   [b2   (op)  (let ((v2 (cdr (sr 0))) (v1 (cdr (sr 1))))
                  (retbaseval (op v1 v2) s (- p 2)))]
   [ret  ()    (retbaseval e s p)]
   [b1   (op)  (let ((v1 (cdr (sr 0))))
                  (retbaseval (op v1) s (- p 1)))]
   [if   ()    (if (cdr (sr 0))
                   (enter (sr 1) s (- p 3))
                   (enter (sr 2) s (- p 3)))]
   [cst  (val) (retbaseval val s p)]
   [pn   ()    (let ((p (updatemrks! t e s p)))
               (match ((sr 0)) by
                   [('(ret) . val) => (print val)  (enter (sr 2) s (- p 3))]
                   [_              => (print "()") (enter (sr 0) s (- p 1))]))]
   [pcr  ()    (let ((u1 (sr 0)))
                  (ss 0 pcp) (ss -1 pnp) (enter u1 s (+ p 1)))]
   [pcp  ()    (let ((p (updatemrks! t e s p)))
               (let ((u1 (sr 0)))
                  (ss 0 pcr) (ss -1 pc) (ss -2 pn) (print " ")
                  (mrk&enter u1 s (+ p 2))))]
   [pc   ()    (let ((p (updatemrks! t e s p)))
               (let ((u1 (sr 0)))
                  (ss 0 pcr) (ss -1 pc) (ss -2 pn) (print "(")
                  (mrk&enter u1 s (+ p 2))))]
   [pnp  ()    (let ((p (updatemrks! t e s p)))
               (match ((sr 0)) by
                   [('(ret) . val) => (print " . " val ")")
                                      (enter (sr 2) s (- p 3))]
                   [_              => (print ")") (enter (sr 0) s (- p 1))]))]
   [letrec (rhss m0) (k m0 (allocm rhss e) s p)]
   [stop   ()  (newline) (println "K machine terminated.") ()]
   [else       (newline) (println "Unknown instruction: " t)]
))

(fun retbaseval (val s p) =
     (let ((p (updatemrks! '(ret) val s p)))
```

```
        (let ((u1 (sr 0)))
          (ss 0 ('(ret) :: val)) (enter u1 s p))))

; Function enter (obj s) enters the closure obj; function mrk&enter in
; addition pushes a marker (pointing to obj) on the stack

(fun enter (obj s p) = (k (car obj) (cdr obj) s p))

(fun mrk&enter (obj s p) =
     (if *doprof* (prof! '(mark)))
     (if *dotrace* (print "marked "))
     (ss -1 ('mrk :: obj))
     (k (car obj) (cdr obj) s (+ p 1)))
```

B.1.5 Updating suspensions in the K machine

Some auxiliary functions for the K machine (and for the K+ machine shown later) are given in the file `kaux.ss` listed below. In particular it contains the functions for updating suspensions and the mechanism for allocating recursive environments. Both makes use of destructive updates of data structures and hence the possibility of creating cyclic structures.

It is essential to note that the run time environment data structure in the K machine refers to nested environment using pointers, not to copies of these environments. Otherwise updating of suspensions and creation of cyclic environments for `letrec` would not achieve lazy evaluation.

Function `updatemrk!` takes three arguments: a piece `m` of code, a pointer `e` to an environment, and a pointer `obj` to some closure `C`•E. It destructively overwrites `C` with `m` and E with `e`. This side effect is the intended effect of `updatemrk!`.

Function `allocm` creates a cyclic run time environment for evaluation of local recursive bindings, as described in Section 3.6. The core L source term (`letrec` $\mathtt{f}_1$=$\mathtt{m}_1$ ... $\mathtt{f}_n$=$\mathtt{m}_n$ `in` $\mathtt{e}_0$) has been compiled to the K instruction (`Lrc` ($\mathtt{m}_1$ ... $\mathtt{m}_n$) $\mathtt{m}_0$). Function `allocm` takes the list $\mathtt{m}_1$ ... $\mathtt{m}_n$ and the current environment `env`, and creates a new cyclic environment E of form E = $\mathtt{m}_1$•E : ... : $\mathtt{m}_n$•E : `env`. The cyclic environment is created by first allocating a "hole" `()` in every new closure in the environment, then destructively overwriting this hole with the new environment.

Finally, the file contains functions for profiling, residency reporting, and tracing the execution of the K machine.

```
; File "kaux.ss" -- Peter Sestoft, 1991-06-24

; Auxiliary functions common to the K and K+ machines

; Stack handling

(extend-syntax (sr) [(sr index) (vector-ref s (- p index))])
(extend-syntax (ss) [(ss index value) (vector-set! s (- p index) value)])

; RUN AND TIME THE MACHINE

(fun run&time (f term) =
```

```
      (initprofile!)
      (collect)
      (let ((s (make-vector 100000 ()))
            (p -1)
            (cells (bytes-allocated)))
         (if *doheap*
             (set! *collect-request-handler*
                   (lambda () (collect)
                           (print " [" (- (bytes-allocated) cells) "] "))))
         (ss -1 stop) (ss -2 pc) (ss -3 pn)
         (time (begin (f term () s (+ p 3)) (*collect-request-handler*)))
         (reset-crh))
      (if *doprof* (printprofile)))

; Function updatemrks! updates all markers on the stack top with the
; object (m . e), then returns the part of the stack below the markers.

(fun updatemrks! (m e s p) =
     (match ((sr 0)) by
            [('mrk . obj) => (updatemrk! m e obj)
                             (updatemrks! m e s (- p 1))]
            [_            => (if *doprof* (prof! '(checkmrk))) p]))

(fun updatemrk! (m e obj) =
     (if *doprof* (prof! '(updatemrk)))
     (if *dotrace* (print "updated "))
     (set-car! obj m) (set-cdr! obj e))

; Function allocm (rhss env) extends environment env to envrec which
; has recursive bindings (mn . envrec): ... :(m1 . envrec):env for all
; terms m in rhss = (m1 ... mn).

(fun allocm (rhss env)
     [()      _ => env]
     [(m . r) _ => (let* ((mclosure (m :: ()))
                          (newe (allocm r (mclosure :: env))))
                     (set-cdr! mclosure newe)
                     newe)])

; Functions for stopping and printing

(define stop '((stop) . ()))
(define pn   '((pn)   . ()))
(define pc   '((pc)   . ()))
(define pnp  '((pnp)  . ()))
(define pcp  '((pcp)  . ()))
(define pcr  '((pcr)  . ()))

; COLLECTING AND PRINTING INSTRUCTION PROFILING INFORMATION

(fun doprof () = (set! *doprof* #t))
(fun noprof () = (set! *doprof* #f))
(fun initprofile! () = (set! *profile* ()))

(fun prof! (inst)
     [(op . _)  =>
     (match ((assq op *profile*)) by
            [(ptr = (_ . count)) => (set-cdr! ptr (+ 1 count))]
            [#f                  => (set! *profile* ((op :: 1) :: *profile*))]
)])

(fun printprofile () =
     (fun prt (entry)
          [(op . count) => (print op)
                           (tabulate (- 15 (symlength op)))
                           (println count)])
```

```
    (fun symlength (sym) = (string-length (symbol->string sym)))
    (fun tabulate (n)
         ['0 => ()]
         [_  => (print " ") (tabulate (- n 1))])
    (map prt (sort (lambda (ent1 ent2) (> (cdr ent1) (cdr ent2))) *profile*))
    ())

; Facilities for tracing K machine instructions

(fun dotrace () = (set! *dotrace* #t))
(fun notrace () = (set! *dotrace* #f))

; Facilities for measuring residency (heap consumption)

(fun reset-crh () = (set! *collect-request-handler* (lambda () (collect))))
(fun doheap () = (set! *doheap* #t))
(fun noheap () = (set! *doheap* #f))
```

The file below contains general auxiliary functions, used by the L implementation as well as by the closure and usage count analyses.

```
; File "auxiliary.ss" -- Peter Sestoft, 1991-06-18

; General auxiliary functions common to all programs

; Strict fold left
(fun sfoldl (f a xs)
     [_ _ ()      => a]
     [_ _ (x . r) => (sfoldl f (f a x) r)])

; Print functions
(define (print . args)
   (map display args)
   (flush-output-port))

(define (println . args)
   (map display args)
   (newline)
   (flush-output-port))


; Four versions of strict "or" (binary; multiarguments; on a list of
; booleans, and iterating a predicate on a list).

(fun sor2 (a1 a2)     = (or a1 a2))
(define (sor a1 . ar) = (sfoldl sor2 a1 ar))
(fun sorl (as)        = (sfoldl sor2 #f as))
(fun sorall (as f)    = (sfoldl (lambda (chg? a) (sor2 chg? (f a))) #f as))
```

B.2 The closure analysis

This section describes the implementation of the closure analysis (Chapter 4). The closure analysis (and the usage count analysis described below) work on decorated core L expressions, described in Section B.2.2.

The analysis is organized in two files: `closure.ss` which contains the closure analysis functions proper, and `aaux.ss` which contains auxiliary functions common to the closure analysis and the usage count analysis.

B.2.1 How to do closure analysis of an L program

To apply the closure analysis to L program `e`, load file `loadkplus` and execute `(closurea e)`. This will print `e` as a decorated core L expression, where the rho and phi properties have been computed by the closure analysis. This print format is not easy to read for large programs and is intended mainly for debugging.

Example 14 Closure analysis for the program from Example 1 in Chapter 5:

```
> (closurea '(letrec (f x y     = (+ x y))
                     (twice g z = (g (g z)))
                     in (twice (f 7) 9)))

(letrec ([f =
          ((letbnd () (5) (many . zero) -)
           (lam (5 (4) () (many . zero) (many . zero))
                x
                (lam (4 () () (many . zero) (many . zero))
                     y
                     (bapp #<procedure +> (var x) (var y))))))]
         [twice =
          ((letbnd () (7) (many . zero) -)
           (lam (7 (6) (4) (many . zero) (many . zero))
                g
                (lam (6 () () (many . zero) (many . zero))
                     z
                     (app (var g) (app (var g) (var z))))))])
   (app (app (var twice) (app (var f) (bapp 7))) (bapp 9)))
```

The output is given in the concrete syntax of decorated core L. In a term such as `(lam (7 (6) (4) ...) ...)`, `7` is the label of the lambda, the phi-list `(6)` shows that it can evaluate only to the lambda labelled `6`, and the rho-list `(4)` shows that it can be applied only the lambda with label `4`. The usage intervals `(many . zero)` concern only the usage interval analysis and are meaningless here.

The result shows that variables `x`, `y`, and `z` cannot be bound to any lambdas and that variable `g` can be bound only to lambda `4`, that is, a partial application of `f` to one argument. It also shows that neither `f` nor `twice` can return a lambda. □

```
; File "closure.ss" -- Peter Sestoft 1991-02-01, 1991-06-17
; Uses "auxiliary", "aaux.ss"

; CLOSURE ANALYSIS OF CORE L PROGRAMS

; The analysis compiles an L program to core L, decorates the program
; with fields for recording analysis information, and applies the
; closure analysis functions pe and pv.
```

```
(fun closurea (e) =
      (set! t (decorate (l2core (addcontext e))))
      (subcontext (iter-pv t)))


; THE CLOSURE ANALYSIS FUNCTIONS

; Function pv is iterated until a fixed point is reached, that is, until
; the annotations stabilize.  The result will often be circular and can
; be printed only with prte.

(fun iter-pv (e) =
     (if (pv e) (iter-pv e) (prte e)))

; The closure analysis function pe computes the set of closures
; (lambdas) to which an expression may evaluate.

(fun pe (e env)
     [('var x)          _ => (get rho (getlab x env))]
     [(('lam x m) . ps) _ => (list e)]
     [('app m n)        _ => (sfoldl (lambda (s0 l) (union s0 (get phi l)))
                                     empty (pe m env))]
     [('if p m n)       _ => (union (pe m env) (pe n env))]
     [('bapp bop . es)  _ => empty]
     [('letrec defs e0) _ => (let ((newe (adddefnames defs env)))
                               (pe e0 newe))])

; The closure propagation function pv pushes the phi and rho information
; further in the expression by analysing all places where binding
; takes place: lambdas, applications, and letrec-bindings.  The
; function returns #t if any new propagation took place, #f otherwise
; (that is, when the information has stabilized).

(fun pv (e) =
     (fun pv (e env)
          [('var x)          _ => #f]
          [(('lam x m) . ps) _ => (let ((newe (var x e env)))
                                     (sor (updset phi e (pe m newe))
                                          (pv m newe)))]
          [('app m n)        _ =>
                     (sor (sorall (pe m env)
                                  (lambda (l) (updset rho l (pe n env))))
                          (pv m env)
                          (pv n env))]
          [('if p m n)       _ => (sor (pv p env) (pv m env) (pv n env))]
          [('bapp bop . es)  _ => (sorall es (lambda (e) (pv e env)))]
          [('letrec defs e0) _ =>
                (let ((newe (adddefnames defs env)))
                   (sor (sorall defs (lambda (def) (pvdef def newe)))
                        (pv e0 newe)))])
     (fun pvdef (def env)
          [(f '= (rhs = (e . ps))) _ => (sor (updset rho rhs (pe e env))
                                             (pv e env))])
     (pv e ()))
```

B.2.2 Decorated core L

The descriptions ϕ and ρ used by the closure analysis, and the descriptions ψ and χ used by the usage count analysis, are represented as decorations on the abstract core L syntax tree. The details of this are described in comments in the file below which lists functions for constructing (`decorate`) and printing (`prte`) decorated core L expressions, as well as for accessing and updating the descriptions (called properties in the file).

```
; File "aaux.ss" -- Peter Sestoft, 1991-06-21
; Uses "auxiliary"

; Auxiliary functions common to the closure and usage count analyses

; DECORATED CORE L EXPRESSIONS

; A decorated core L program has a list of properties (attributes) for
; each lambda and letrec binding in the syntax tree.  The properties are
; called inx, phi, rho, psi, and chi.  To access the value of property p
; of node n in the tree one executes (get p n); to set property p of n
; to value val, one executes (put p n val); to transform property p of
; node n by function f, one executes (upd p n f).  This returns #t if
; the values was actually changed (according to eq?), #f otherwise.
;
; WARNING: Properties are modified destructively by the analyses, and
; should be accessed only using the functions get, put, and upd, and the
; properties inx, phi, rho, psi, chi.
;
; The properties are added by replacing the subtree at node expr by a
; pair (expr1 . properties), where expr1 is the result of applying this
; process recursively to all subtrees of expr.

; Representation of decorated core L expressions
; <dex> ::= ((lam <var> <dex>) . <properties>)
;         | (var <sym>)
;         | (app <dex> <dex>)
;         | (bapp <baseop> <dex>*)
;         | (if <dex> <dex> <dex>)
;         | (letrec <ddf>* <dex>)
; <ddf> ::= (<var> = (<dex> . <properties>))
; <properties> ::= (<inx> <phi> <rho> <psi> <chi>)

; For a lambda l the properties are interpreted as follows:
; inx l = the unique printable label (a number) of l
; phi l = the set of lambdas that l can evaluate to when applied
; rho l = the set of lambdas that l can be applied to (= its var. bound to)
; psi l = the usage interval of l's variable
; chi l = the usage interval of l itself
;
; For a let-bound variable f only the following properties are used:
; rho f = the set of lambdas that f can be bound to
; psi f = the usage interval of f

; DECORATING CORE L EXPRESSIONS

(fun decorate (e) =
     (define occurrence 0)
     (fun newocc () = (set! occurrence (+ occurrence 1)) (list occurrence))
     (fun allocp (e)
          [('var x)          => e]
          [('lam x m)        => (addlamprops `(lam ,x ,(allocp m)) newocc)]
          [('app m n)        => `(app ,(allocp m) ,(allocp n))]
          [('if p m n)       => `(if ,(allocp p) ,(allocp m) ,(allocp n))]
          [('bapp bop . es) => ('bapp :: bop :: (map allocp es))]
          [('letrec es e0)  => `(letrec ,(map allocdef es) ,(allocp e0))])
     (fun allocdef (def)
          [(f '= rhs) =>  `(,f = ,(addletprops (allocp rhs)))])
     (allocp e))

; Add properties to a node: It is important that a cons cell is created
```

```
; anew at every property (the references created must be distinct).

(fun addlamprops (e newocc) =
     (define (emptyset) (list empty))
     (define (emptyint) (list (emptyui)))
     (e :: (list (newocc) (emptyset) (emptyset) (emptyint) (emptyint) )))
; Properties:      inx      phi        rho        psi        chi

(fun addletprops (e) =
     (define (emptyset) (list empty))
     (define (emptyint) (list (emptyui)))
     (e :: (list '(letbnd) '(-)       (emptyset) (emptyint) '(-) )))

; ACCESSING PROPERTIES

(fun upd (property node transform) =
     (let* ((ptr      (property (cdr node)))
            (value    (transform (car ptr)))
            (changed? (not (eq? value (car ptr)))))
        (set-car! ptr value)
        changed?))

(fun get (property node) = (car (property (cdr node))))

(fun put (property node value) = (set-car! (property (cdr node)) value))

(fun inx (properties) = (car properties))
(fun phi (properties) = (car (cdr properties)))
(fun rho (properties) = (car (cdr (cdr properties))))
(fun psi (properties) = (car (cdr (cdr (cdr properties)))))
(fun chi (properties) = (car (cdr (cdr (cdr (cdr properties))))))

; PRINTING DECORATED CORE L EXPRESSIONS

; The properties are printed in the following notation:
; The lambdas are numbered for identification, and this number is
; printed ahead of the properties:
; (lam <number> (<phi> <rho> <psi> <chi>) <var> <body>)
; The letrec bindings have no numbers but are marked (letbnd) instead:
; (letrec <fun> = (((letbnd) (<phi> (-) <psi> (-)) <body>) ...) <body>)

(fun prte (e)
     [('var x)          => e]
     [(('lam x m) . ps) => `(lam ,(printprops e) ,x ,(prte m))]
     [('app m n)        => `(app ,(prte m) ,(prte n))]
     [('if p m n)       => `(if ,(prte p) ,(prte m) ,(prte n))]
     [('bapp bop . es)  => ('bapp :: bop :: (map prte es))]
     [('letrec defs e0) => `(letrec ,(map prtdef defs) ,(prte e0))])

(fun prtdef (def)
     [(f '= (rhs = (e . ps))) => `(,f = ,(list (printprops rhs) (prte e)))])

(fun printprops (node) =
     (list (get inx node) (prtset phi node) (prtset rho node)
           (prtintv psi node) (prtintv chi node)))

; Printing a set property using the unique lambda labels:

(fun prtset (prop node) =
     (map (lambda (e) (get inx e)) (get prop node)))

; Printing an interval property

(fun prtintv (prop node) = (get prop node))

; THE ANALYSIS TIME ENVIRONMENT
```

```
; An environment (as used by the closure and usage count analyses) is
; a list of triples (x . (label .  whnf?)) where x is the name of a
; variable, label is the label of its binding lambda or letrec, and
; whnf? is #t if the variable will only be bound to closures in WHNF, #f
; otherwise.

(fun var  (x e env) = ((x :: (e :: #f)) :: env))

(fun var# (x e env) = ((x :: (e :: #t)) :: env))

; Getting the binding label of a variable
(fun getlab (var env) =
   (match ((assoc var env)) by
      [(_ . (lab . whnf?)) => lab]
      [#f                  => (error 'getlab "Unknown variable: ~s" var)]))

(fun adddefnames (defs env) =
     (fun adddefname (env def)
          [_ (f '= (rhs = ( (('lam _ _) . _) . _))) => (var# f rhs env)]
          [_ (f '= rhs)                             => (var  f rhs env)])
     (sfoldl adddefname env defs))

; Functions for simulating the context given by the printer

(fun addcontext (e) =
     `(letrec (prtnil = (lam z z))
              (prtcons x y = (+ (x prtnil prtcons)
                                (y prtnil prtcons)))
              in (,e prtnil prtcons)))

(fun subcontext (e)
     [('letrec (prtnil prtcons) (app (app e _) _)) => e])

; Adding a set of occurrences to a property.  Only those elements of
; the newset which are not in the oldset already will be added.  The
; function returns #t if any new element was added, #f otherwise.  Note that
; it is important that (union s1 s2) will return s2 eq?-unmodified if no new
; element was added from s1.

(fun updset (prop node newset) =
     (upd prop node (lambda (oldset) (union newset oldset))))

(fun union (s1 s2)
     [(v1 . vr) _ => (if (memv v1 s2)
                         (union vr s2)
                         (v1 :: (union vr s2)))]
     [()        _ => s2])

(define empty ())                       ; The empty set
(fun empty? (s) = (s =? empty))

(fun makeui (ui1 ui2) = (ui1 :: ui2))
(define (oneui)   (makeui 'one  'one))    ; Unit for uimul2
(define (zeroui)  (makeui 'zero 'zero))   ; Unit for uiadd2
(define (emptyui) (makeui 'many 'zero))   ; Unit for uilub2
(define (manyui)  (makeui 'zero 'many))
```

B.3 The usage count analysis

This section lists the implementation of the usage count analysis, the optimizing code generation, and the K+ machine which executes the optimized code. (Chapter 5).

The usage count analysis works on decorated core L programs, as described in the section on closure analysis above.

B.3.1 How to do usage count analysis for L programs

To apply the usage count analysis to L program `e`, load file `loadkplus` and execute `(usagea e)`. This will print `e` as a decorated core L expression, where the psi and xi properties (and the rho and phi properties of the closure analysis) have been computed by the usage count analysis.

Example 15 Usage interval analysis for the program from Example 1 in Chapter 5:

```
> (usagea '(letrec (f x y = (+ x y))
                   (twice g z = (g (g z)))
                   in (twice (f 7) 9)))
(letrec ([f =
          ((letbnd () (5) (many . many) -)
           (lam (5 (4) () (one . many) (one . many))
                x
                (lam (4 () () (one . one) (one . many))
                     y
                     (bapp #<procedure +> (var x) (var y)))))]
         [twice =
          ((letbnd () (7) (one . one) -)
           (lam (7 (6) (4) (many . many) (one . one))
                g
                (lam (6 () () (one . one) (one . one))
                     z
                     (app (var g) (app (var g) (var z))))))])
   (app (app (var twice) (app (var f) (bapp 7))) (bapp 9)))
```

The result is shown in decorated core L syntax, and contains information from the closure analysis as well as from the usage interval analysis. In the lambda `(lam (7 (6) (4) (many . many) (one . one)) ...)`, the psi-interval `(many . many)` is the usage interval of the variable (here `g`) bound at the lambda, and the chi-interval `(one . one)` is the usage interval of the lambda itself.

The result shows that `x` has interval $[One, Many]$, `y` has $[One, One]$, `g` has $[Many, Many]$, and `z` has $[One, One]$. The only sharable lambdas are `4` and `5`. □

The file below contains the usage count analysis, which relies on the auxiliary functions in file `aaux.ss` listed in Section B.2.2 above.

```
; File "usagecount.ss" -- Peter Sestoft 1991-01-31, 1991-06-21
; Uses "auxiliary", "compile", "aaux", "closure"

; USAGE INTERVAL ANALYSIS OF CORE L PROGRAMS
```

```
(fun usagea (e) =
     (closurea e)
     (iter-uv t)
     (prte (subcontext t)))

; The application (iter-uv e) will compute usage intervals for all
; lambdas and variables in the program.

(fun iter-uv (e) =
     (if (uv e) (iter-uv e) (prte e)))

; Computing the usage interval of a variable in an expression.
; The usage interval for a variable in an expression is computed by
; the function ue, which therefore plays a role analogous to that of
; function pe in the closure analysis.  The application (ue e env v)
; returns the usage interval of variable v in expression e, given that
; expression e is used once.

(fun ue (e env v)
     [('var x)          _ _ => (if (x =? v) (oneui) (zeroui))]
     [(('lam x m) . ps) _ _ => (if (x =? v)
                                   (zeroui)
                                   (uimul2 (get chi e)
                                           (ue m (var x e env) v)))]
     [('app m n)        _ _ => (uiadd2 (ue m env v)
                                       (uimul2 (ue n env v)
                                               (psilub m env)))]
     [('if p m n)       _ _ => (uiadd2 (ue p env v)
                                       (uilub2 (ue m env v) (ue n env v)))]
     [('bapp bop . es)  _ _ => (uiaddl (map (lambda (e) (ue e env v)) es))]
     [('letrec defs e0) _ _ =>  (if (member v (map car defs))
                                    (zeroui)
                                    (let ((newe (adddefnames defs env)))
                                       (ueletrec defs e0 newe v)))])

(fun ueletrec (defs e0 env v) =
     (fun uedef (def)
          [(f '= (rhs = (e . ps))) => (uimul2 (get psi rhs) (ue e env v))])
     (uiadd2 (uiaddl (map uedef defs)) (ue e0 env v)))

; Propagating the usage interval information.
; The usage interval information is propagated in the expression using
; function uv, which is analogous to the pv function in the closure
; analysis.

(fun uv (e) =
     (fun uv (e env)
          [('var x)          _ => #f]
          [(('lam x m) . ps) _ => (let ((newe (var x e env)))
                                      (sor (updui psi e (ue m newe x))
                                           (updchi (get rho e) (get psi e))
                                           (uv m newe)))]
          [('app m n)        _ => (sor (uv m env)
                                       (uv n env)
                                       (updchi (pe m env) (makeui 'one 'one)))]
          [('if p m n)       _ => (sor (uv p env) (uv m env) (uv n env))]
          [('bapp bop . es)  _ => (sorall es (lambda (e) (uv e env)))]
          [('letrec defs e0) _ =>
                (let ((newe (adddefnames defs env)))
                   (fun uvdef (def)
                        [(f '= (rhs = (e . ps))) =>
                            (sor (updchi (get rho rhs) (get psi rhs))
                                 (updui psi rhs (ueletrec defs e0 newe f))
                                 (uv e newe))])
                   (sor (sorall defs uvdef) (uv e0 newe)))])
```

```
        (uv e ()))

    ; For every lambda l in the set labs, update the chi of l with the
    ; lub of it and the usage interval of e.

    (fun updchi (labs ui) =
        (sorall labs (lambda (lab) (updui chi lab ui))))

    ; (psilub m env) is the lub of the usage intervals of the lambdas that
    ; m can evaluate to:

    (fun psilub (m env) =
        (uilubl (map (lambda (l) (get psi l)) (pe m env))))

    ; USAGE COUNTS, USAGE INTERVALS, AND OPERATIONS ON THEM.

    ; A usage count is zero, one, or many.
    ; A usage interval is a pair of usage counts.

    (fun ucmin2 (uc1 uc2)
        ['zero _     => uc1]
        [_     'many => uc1]
        [_     _     => uc2])

    (fun ucmax2 (uc1 uc2)
        ['zero _     => uc2]
        [_     'many => uc2]
        [_     _     => uc1])

    (fun ucadd2 (uc1 uc2)
        ['zero _     => uc2]
        [_     'zero => uc1]
        [_     _     => 'many])

    (fun ucmul2 (uc1 uc2)
        ['zero  _    => uc1]
        [_     'one  => uc1]
        [_      _    => uc2])

    (fun uiadd2 (ui1 ui2)
        [(u11 . u12) (u21 . u22) => ((ucadd2 u11 u21) :: (ucadd2 u12 u22))])
    (fun uiaddl (uis) = (sfoldl uiadd2 (zeroui) uis))

    (fun uimul2 (ui1 ui2)
        [(u11 . u12) (u21 . u22) => ((ucmul2 u11 u21) :: (ucmul2 u12 u22))])

    (fun uilub2 (ui1 ui2)
        [(u11 . u12) (u21 . u22) => ((ucmin2 u11 u21) :: (ucmax2 u12 u22))])
    (fun uilubl (uis) = (sfoldl uilub2 (emptyui) uis))

    (fun upperb (ui1) [(u11 . u12) => u12])
    (fun lowerb (ui1) [(u11 . u12) => u11])

    (fun updui (prop node newui) =
        (upd prop node (lambda (oldui) (let ((resui (uilub2 oldui newui)))
                                          (if (resui =? oldui) oldui resui)))))
```

B.4 Optimized compilation and the K+ machine

Finally we describe the optimized compilation of L programs, and the extended K machine, called the K+ machine, which executes the code generated.

B.4.1 How to optimize and run L programs

To optimize and run L program e, load file `loadkplus` and execute `(k+run e)`. This will apply closure analysis and usage count analysis, do optimized compilation into K+ machine code, and run the K+ machine on the optimized code. Otherwise it behaves exactly as the K machine, so profiling, residency reporting, and tracing can be used on this machine as well (Section B.1.2).

The file below contains the optimized compilation and the K+ machine interpreter. It relies on the auxiliary functions in file `kaux.ss` listed in Section B.1.5.

```
; File "kplusmachine.ss" -- Peter Sestoft, 1991-06-19, 1991-09-30
; Uses "auxiliary", "compile", "closure", "usagecount", "kaux"

; Interpretive implementation of the K+ machine (a lazy Krivine machine).
; This is the K machine extended with optimized instructions.

; The K+ instructions are as follows:
; (These instructions are named as in Chapter 6 of the report).
; <ins> ::= (lam# <ins>)                Unsharable abstraction
;         | (sla  <ins>)                Strict lambda
;         | (sla# <ins>)                Unsharable strict lambda
;         | (var# <num>)                Unsharable variable
;         | (app# <ins> <num>)          Application to unsharable variable
;         | ... the usual K machine instructions

; The data structures are as for the basic K machine.

; MAIN FUNCTIONS TO COMPILE, OPTIMIZE, RUN, AND TIME PROGRAMS
; -----------------------------------------------------------

(fun k+run (exp) =
    (run&time k+ (k+comp exp)))

(fun k+comp (e) =
     (usagea e)
     (dc2k+ (subcontext t)))

; COMPILATION FROM DECORATED CORE L TO K+ INSTRUCTIONS

; The usage interval and closure information previously computed can be
; exploited in code generation, using special K+ machine instructions such
; as var# (unsharable variable), lam# (unsharable lambda), app# (application
; to unsharable variables).

(fun dc2k+ (e) =
  (fun gen (e env) = ; (print (car e))
  (match (e) by
     [(('lam x m). ps) =>
        (match ((lowerb (get psi e)) (upperb (get chi e))) by
       ['zero 'many                     => `(lam  ,(gen m (var  x e env)))]
       ['zero _                         => `(lam# ,(gen m (var  x e env)))]
       [_    'many & (empty? (get rho e))  => `(sla  ,(gen m (var# x e env)))]
       [_    _ & (empty? (get rho e))      => `(sla# ,(gen m (var# x e env)))]
       [_     'many                     => `(lam  ,(gen m (var  x e env)))]
       [_     _                         => `(lam# ,(gen m (var  x e env)))]
        )]
     [('var x)         => (match ((upperb (get psi (getlab x env)))) by
                             ['many => (getind x env 'var)]
                             [_     => (getind x env 'var#)])]
     [('app m n)       => (mkapp# (gen m env) (gen n env))]
     [('bapp c)        => `(cst ,c)]
```

```
      [('bapp op e1)    => `(app ,(gen e1 env) (b1 ,op))]
      [('bapp op e1 e2) => `(app (app (app ,(gen e1 env) (r1))
                                      ,(gen e2 env)) (b2 ,op))]
      [('if p m n)      => `(app (app (app ,(gen p env) (if))
                                      ,(gen m env)) ,(gen n env))]
      [('letrec defs e0)=>
              (let ((newe (adddefnames defs env)))
                 `(letrec ,(map (lambda (def) (gendef def newe)) defs)
                     ,(gen e0 newe)))]
      [_                => (error 'dc2k+ "Unknown expression: ~s" e)]))
      (fun gendef (def env)
           [(f '= (e . ps)) _ => (gen e env)])
      (gen e ()))

; Variable lookup in the analysis environment

(fun getind (v env ins) =
     (fun get-aux (env index)
          [()       _ => (error 'dc2k+ "Undefined variable: ~s " v)]
          [((v1 . (_ . tag)) . r) _ & (v1 =? v) =>
                                (if tag
                                    `(var#  ,index)
                                    `(,ins  ,index))]
          [(_ . r) _ => (get-aux r (+ 1 index))])
     (get-aux env 0))

(fun mkapp# (m n)
     [_ ('var# i) => `(app# ,m ,i)]
     [_ _         => `(app  ,m ,n)])

; THE K+ MACHINE INTERPRETATION FUNCTION
; -------------------------------------

(fun k+ (t e s p) =
  (if *doprof* (prof! t))
  (if *dotrace* (print (car t) " "))
  (record-case t
   [app  (m n) (ss -1 (n :: e)) (k+ m e s (+ 1 p))]
   [app# (m i) (ss -1 (list-ref e i)) (k+ m e s (+ 1 p))]
   [var  (i)   (k+mrk&enter (list-ref e i) s p)]
   [var# (i)   (k+enter     (list-ref e i) s p)]
   [lam  (m)   (let ((p (updatemrks! t e s p)))
                  (k+ m ((sr 0) :: e) s (- p 1)))]
   [lam# (m)   (k+ m ((sr 0) :: e) s (- p 1))]        ; (sr 0) is not a marker
   [sla  (m)   (let ((p (updatemrks! t e s p)))
               (let ((u1 (sr 0)))
                  (ss 0 (`(lam# ,m) :: e))
                  (k+enter u1 s p)))]
   [sla# (m)   (let ((u1 (sr 0)))                     ; (sr 0) is not a marker
                  (ss 0 (`(lam# ,m) :: e))
                  (k+enter u1 s p))]
   [r1   ()    (let ((u1 (sr 0)) (u2 (sr 1)) (uo (sr 2)))
                  (ss 2 u1) (ss 1 uo) (k+enter u2 s (- p 1)))]
   [b2   (op)  (let ((v2 (cdr (sr 0))) (v1 (cdr (sr 1))))
                  (k+retbaseval (op v1 v2) s (- p 2)))]
   [ret  ()    (k+retbaseval e s p)]
   [b1   (op)  (let ((v1 (cdr (sr 0))))
                  (k+retbaseval (op v1) s (- p 1)))]
   [if   ()    (if (cdr (sr 0))
                   (k+enter (sr 1) s (- p 3))
                   (k+enter (sr 2) s (- p 3)))]
   [cst  (val) (k+retbaseval val s p)]
   [pn   ()    (let ((p (updatemrks! t e s p)))
               (match ((sr 0)) by
                  [('(ret). val) => (print val)  (k+enter (sr 2) s (- p 3))]
                  [_             => (print "()") (k+enter (sr 0) s (- p 1))]))]
```

```
    [pcr  ()    (let ((u1 (sr 0)))
                   (ss 0 pcp) (ss -1 pnp) (k+enter u1 s (+ p 1)))]
    [pcp  ()    (let ((p (updatemrks! t e s p)))
                (let ((u1 (sr 0)))
                   (ss 0 pcr) (ss -1 pc) (ss -2 pn) (print " ")
                   (k+mrk&enter u1 s (+ p 2))))]
    [pc   ()    (let ((p (updatemrks! t e s p)))
                (let ((u1 (sr 0)))
                   (ss 0 pcr) (ss -1 pc) (ss -2 pn) (print "(")
                   (k+mrk&enter u1 s (+ p 2))))]
    [pnp  ()    (let ((p (updatemrks! t e s p)))
                (match ((sr 0)) by
                    [('(ret). val) => (print " . " val ")")
                                         (k+enter (sr 2) s (- p 3))]
                    [_             => (print ")") (k+enter (sr 0) s (- p 1))])))]
    [letrec (rhss m0) (k+ m0 (allocm rhss e) s p)]
    [stop   ()  (newline) (println "K plus machine terminated.") ()]
    [else       (newline) (println "Unknown instruction: " t)]
))

(fun k+retbaseval (val s p) =
     (let ((p (updatemrks! '(ret) val s p)))
        (let ((u1 (sr 0)))
           (ss 0 ('(ret) :: val)) (k+enter u1 s p))))

; Function k+enter (obj s) enters the closure obj; function k+mrk&enter in
; addition pushes a marker (pointing to obj) on the stack

(fun k+enter (obj s p) = (k+ (car obj) (cdr obj) s p))

(fun k+mrk&enter (obj s p) =
     (if *doprof* (prof! '(mark)))
     (if *dotrace* (print "marked "))
     (ss -1 ('mrk :: obj))
     (k+ (car obj) (cdr obj) s (+ p 1)))
```

B.5 Benchmark programs in L

These are the L functions used in the benchmark runs in Section 5.4.1.

```
; File "examples" -- Peter Sestoft 1991-01-21
; Examples of L source terms for benchmarks

(define fact '(fact n = (if (= 0 n) 1 (* n (fact (- n 1))))))

(define ones '(ones = (:: 1 ones)))

(define repeat
   '(repeat x = (xs where (xs = (:: x xs)))))

(define from '(from n = (:: n (from (+ n 1)))))

(define nth
   '(nth n l = (case l of
                  () => ()
                  (:: x r) => (if (= n 0) x (nth (- n 1) r)))))

(define drop
   '(drop n xs = (case xs of
                    () => ()
```

```
                      (:: x r) => (if (= 0 n) xs (drop (- n 1) r)))))

(define take
   '(take n xs = (case xs of
                      () => ()
                      (:: x r) => (if (= 0 n)
                                      ()
                                      (:: x (take (- n 1) r))))))

(define filter
   `(filter p l = (case l of
                       () => ()
                       (:: x r) => (if (p x)
                                       (:: x (filter p r))
                                       (filter p r)))))
(define foldl
   '(foldl g z l = (case l of
                        () => z
                        (:: x r) => (foldl g (g z x) r))))

(define scan
   '(scan g a xs = (:: a (case xs of
                                 () => ()
                                 (:: x xs) => (scan g (g a x) xs)))))

; This version of mapzip2 works for infinite lists only:

(define mapzip2
   '(mapzip2 g xs ys = (:: (g (hd xs) (hd ys))
                           (mapzip2 g (tl xs) (tl ys)))))

(define fibs `(fibs = (letrec ,mapzip2 in
                              (:: 1 (:: 1 (mapzip2 + (tl fibs) fibs))))))
(define sieve
   '(sieve l = (case l of
                    () => ()
                    (:: a r) =>
                    (:: a (sieve (filter (lam x (not (= 0 (remainder x a))))
                                         r))))))

(define primes `(primes = (letrec ,filter ,sieve ,from in (sieve (from 2)))))
```

Appendix C

Dansk Sammenfatning

Denne afhandling beskriver automatisk analyse og transformation af såkaldte "rene" funktionsprogrammer (dvs. evaluering foregår uden sideeffekter). Formålet med en programanalyse er at opsamle information om et givet program, sædvanligvis for at støtte en transformation af programmet. Formålet med programtransformation er at optimere, dvs. forbedre, et programs implementation ved at reducere dets lager- eller køretidsforbrug.

Behovet for sådanne optimerende transformationer er særlig stort for funktionssprog, idet sprogfraser kan kombineres på meget generel måde (vha. højereordens funktioner, dovne datastrukturer, mv.). Oversætter man fraserne enkeltvis, må hver frase implementeres med en generel og bekostelig mekanisme. I et konkret program bruges hver enkelt forekomst af en frase dog ofte på en særlig simpel måde, og kan derfor implementeres med en speciel og mindre bekostelig mekanisme.

Automatisk programanalyse kan lokalisere sådanne simple anvendelser, og automatisk programtransformation kan erstatte den generelle mekanisme med en simplere og mindre bekostelig for en given forekomst af en frase i et program. For at analysere hvordan den enkelte frase bruges, må en programanalyse opsamle *operationel* information, dvs. information om hvad der sker med et program og dets enkelte deludtryk når det udføres.

Alle programanalyserne beskrevet i afhandlingen opsamler sådan operationel information. Dette må ske uafhængigt af programmernes inddata for at give information der er nyttig og sikker for alle mulige udførelser. Af beregnelighedsårsager giver analyserne derfor nødvendigvis *approksimativ* (men sikker) information om programmernes køretidsadfærd.

I afhandlingen defineres et eksempelsprog og dets implementation: et dovent højereordens funktionssprog med datastrukturer. Hovedindholdet i afhandlingen er dog en række automatiske programanalyser og tilhørende programtransformationer.

Først en *funktionsanalyse* ("closure analysis") der finder en approksimation til mængden af funktioner der kan anvendes ved hver funktionsanvendelse i programmet. Funktionsanalysen er oprindelig udviklet af forfatteren i anden sammenhæng men er her generaliseret til det mere omfattende eksempelsprog. Funktionsanalysen bruges normalt som hjælpeanalyse til at udvide førsteordensanalyser til højereordensprogrammer. Denne analyse er implementeret og bevist korrekt.

Dernæst beskrives en brugsintervalanalyse ("usage interval analysis") som finder øvre og nedre grænser på antallet af gange et udtryk vil blive evalueret. Den nedre grænse giver strikthedsinformation, og den øvre grænse giver information om deling af beregninger; begge muliggør optimering af parameteroverførsel. Brugsintervalanalysen forudsætter en funktionsanalyse. Det konkluderes at fornuftig analyse af datastrukturer kræver at man gør brug af typestrukturen i et program, hvad denne analyse ikke gør. Analysen er implementeret.

For et førsteordenssprog med dovne datastrukturer præsenteres nu en familie af evalueringsordensanalyser (som udnytter programmets typestruktur). En evalueringsordensanalyse finder approksimativ information om den rækkefølge i hvilken forskellige deludtryk af det samme udtryk evalueres til svag hovednormalform ("weak head normal form") i en given kontekst. Sådanne analyser generaliserer baglæns strikthedsanalyse, og den opsamlede information bruges til at optimere implementationen af parameteroverførsel. Dette er så vidt vides den første analysemetode for evalueringsorden i sprog med dovne datastrukturer. Disse analyser er ikke implementeret.

Endelig præsenteres en globaliseringsanalyse for et strikt højereordens funktionssprog. Globaliseringsanalysen finder for et forelagt program hvilke funktionsparametre der kan erstattes med globale variable. Globaliseringsanalysen udvider tidligere arbejde idet den tillader globalisering af parametre som "indfanges" i partielle funktionsanvendelser ("closures"). Analysen er implementeret og bevist korrekt med hensyn til en abstrakt maskine, nemlig Landins SECD-maskine.

Bibliography

[1] S. Abramsky. Strictness analysis and polymorphic invariance. In H. Ganzinger and N.D. Jones, editors, *Programs As Data Objects, Copenhagen, Denmark, October 1985. (Lecture Notes in Computer Science, vol. 217)*, pages 1–23. Springer-Verlag, 1986.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] G. Argo. Improving the Three Instruction Machine. In *Fourth International Conference on Functional Programming Languages and Computer Architecture. Imperial College, London*, pages 100–112. ACM, Addison-Wesley, September 1989.

[4] G. Argo. *Efficient Laziness.* PhD thesis, Department of Computing Science, University of Glasgow, March 1990. 123+12 pages.

[5] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, revised edition edition, 1984.

[6] A. Bloss. *Path Analysis and the Optimization of Non-Strict Functional Languages.* PhD thesis, Computer Science Department, Yale University, New Haven, Connecticut, USA, May 1989. Also: Research Report YALEU/DCS/RR-704. 129 pages.

[7] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Fourth International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London*, pages 26–38. ACM Press and Addison-Wesley, September 1989.

[8] A. Bloss and P. Hudak. Path semantics. In M. Main et al., editors, *Mathematical Foundations of Programming Language Semantics, 3rd Workshop, New Orleans, Louisiana. (Lecture Notes in Computer Science, vol. 298)*, pages 476–489. Springer-Verlag, 1988.

[9] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1(2):147–164, September 1988.

[10] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[11] G.L. Burn, C.L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. In H. Ganzinger and N.D. Jones, editors, *Programs As Data Objects, Copenhagen, Denmark, October 1985. (Lecture Notes in Computer Science, vol. 217)*, pages 42–62. Springer-Verlag, 1986.

[12] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture, Nancy, France. (Lecture Notes in Computer Science, vol. 201)*, pages 50–64. Springer-Verlag, 1985.

[13] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles on Programming Languages, Los Angeles, California, January 1977*, pages 238–252. ACM, 1977.

[14] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 269–282. ACM, January 1979.

[15] P. Crégut. An abstract machine for the normalization of λ-terms. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 333–340. ACM Press, 1990.

[16] P.L. Curien. The $\lambda\rho$-calculus: an abstract framework for environment machines. Rapport de Recherche LIENS-88-10, Ecole Normale Supérieure, Paris, France, 1988.

[17] M. Draghicescu and S. Purushothaman. A compositional analysis of evaluation-order and its application. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 242–250. ACM Press, 1990.

[18] M. Draghicescu and S. Purushothaman. Static analysis of lazy functional languages. University of Massachusetts at Boston. 28 pages. Submitted to Theoretical Computer Science, 1991.

[19] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[20] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[21] J. Fairbairn. Removing redundant laziness from supercombinators. In L. Augustsson et al., editors, *Implementation of Functional Languages*, pages 181–189, Gothenburg, Sweden, 1985. Programming Methodology Group, Chalmers University of Technology. PMG Report 17.

[22] J. Fairbairn and S.C. Wray. Code generation techniques for functional languages. In *ACM Conf. on LISP and Functional Programming, Cambridge, Massachusetts*, pages 94–104. ACM, 1986.

[23] J. Fairbairn and S.C. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon. (Lecture Notes in Computer Science, vol. 274)*, pages 34–45. Springer-Verlag, 1987.

[24] P. Fradet. Syntactic detection of single-threading using continuations. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 241–258. Springer-Verlag, 1991.

[25] M.R. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[26] M. Georgeff. Transformations and reduction strategies for typed lambda expressions. *ACM Transactions on Programming Languages and Systems*, 6(4):603–631, October 1984.

[27] B. Goldberg. Detecting sharing of partial applications in functional programs. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon. (Lecture Notes in Computer Science, vol. 274)*, pages 408–425. Springer-Verlag, 1987.

[28] B. Goldberg and Y.G. Park. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 152–160. Springer-Verlag, 1990.

[29] C.K. Gomard. *Program Analysis Matters*. PhD thesis, DIKU, University of Copenhagen, Denmark, November 1991. Also DIKU Report 91/17.

[30] C.K. Gomard and P. Sestoft. Evaluation order analysis for lazy data structures. In *Preliminary Proceedings, Fifth Glasgow Functional Programming Workshop, Isle of Skye, Scotland, August 1991*, pages 125–149. Department of Computing Science, University of Glasgow, Scotland, 1991.

[31] C.K. Gomard and P. Sestoft. Globalization and live variables. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 166–177. ACM, 1991.

[32] P. Henderson. *Functional Programming. Application and Implementation*. Prentice-Hall, 1980.

[33] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[34] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 3, pages 45–62. Ellis Horwood, Chichester, England, 1987.

[35] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Thirteenth ACM Symp. Principles of Programming Languages, St. Petersburg, Florida, 1986*, pages 97–109, 1986.

[36] J. Hughes. Strictness detection in non-flat domains. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark. (Lecture Notes in Computer Science, vol. 217)*, pages 112–135. Springer-Verlag, 1986.

[37] J. Hughes. Analysing strictness by abstract interpretation of continuations. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 4, pages 63–102. Ellis Horwood, Chichester, England, 1987.

[38] J. Hughes. Backwards analysis of functional programs. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 187–208. North-Holland, 1988.

[39] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[40] T.P. Jensen and T.Æ. Mogensen. A backwards analysis for compile-time garbage collection. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 227–239. Springer-Verlag, 1990.

[41] T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984. Proc. ACM SIGPLAN '84 Symposium on Compiler Construction.

[42] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture, Nancy, France. (Lecture Notes in Computer Science, vol. 201)*, pages 190–203. Springer-Verlag, 1985.

[43] G. Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87. 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany (Lecture Notes in Computer Science, vol. 247)*, pages 22–39. Springer-Verlag, 1987.

[44] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.

[45] A. Melton, D.A. Schmidt, and G. Strecker. Galois connections and computer science applications. In D. Pitt et al., editors, *Categoty Theory and Computer Programming, Guildford, UK, September 1985. (Lecture Notes in Computer Science, vol. 240)*, pages 299–312. Springer-Verlag, 1986.

[46] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[47] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, Department of Computer Science, University of Edinburgh, Scotland, 1981. 180 pages. Also report CST-15-81.

[48] P. Naur. Checking of operand types in Algol compilers. *BIT*, 5:151–163, 1965.

[49] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.

[50] Y.G. Park and B. Goldberg. Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 178–189. ACM Press, 1991.

[51] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[52] S.L. Peyton Jones. The spineless tagless G-machine: A second attempt. In *Draft Proceedings, Fourth Annual Workshop on Functional Programming, Isle of Skye, Scotland, August 1991*, pages 438–484. Department of Computing Science, University of Glasgow, Scotland, 1991.

[53] S.L. Peyton Jones and J. Salkild. The spineless tagless G-machine. In *Fourth International Conference on Functional Programming Languages and Computer Architecture. Imperial College, London*, pages 184–201. ACM, Addison-Wesley, September 1989.

[54] B. Pierce, S. Dietzen, and S. Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Computer Science Department, Carnegie Mellon University, Pittsburgh, 1989.

[55] G.D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.

[56] J.C. Reynolds. Three approaches to type structure. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Mathematical Foundations of Software Development, TAPSOFT, CAAP'85, Berlin, Germany (Lecture Notes in Computer Science, vol. 185)*, pages 97–138. Springer-Verlag, 1985.

[57] D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):299–310, April 1985.

[58] P. Sestoft. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988.

[59] P. Sestoft. Replacing function parameters by global variables. In *Fourth International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London*, pages 39–53. ACM Press and Addison-Wesley, September 1989.

[60] O. 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.

[61] O. 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.

[62] O. Shivers. The semantics of Scheme control flow analysis (preliminary). School of Computer Science, Carnegie Mellon University. 11 pages, February 1990.

[63] O. 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)*, pages 190–198. ACM, 1991.

[64] M. Sintzoff. Calculating properties of programs by valuations on specific models. *Sigplan Notices*, 7(1):203–207, January 1972. ACM Conference on Proving Assertions about Programs, Las Cruces, Mexico.

[65] H. Søndergaard. *Semantics-Based Analysis and Transformation of Logic Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1989. Also DIKU Report 89/22.

[66] G.L. Steele. Rabbit: A compiler for Scheme. (A study in compiler optimization). Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1978.

[67] B. Steensgaard and M. Marquard. Parameter splitting in a higher order functional language. DIKU Student Project 90-7-1, DIKU, University of Copenhagen, August 1990.

[68] J. Steensgaard-Madsen. Typed representation of objects by functions. *ACM Transactions on Programming Languages and Systems*, 11(1):67–89, January 1989.

[69] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[70] D.A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9:31–49, 1979.

[71] D.A. Turner. The semantic elegance of applicative languages. In *1981 ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, pages 85–92. ACM, 1981.

[72] P. Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis Horwood, Chichester, England, 1987.

[73] P. Wadler and R.J.M. Hughes. Projections for strictness analysis. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture, Portland, Oregon. (Lecture Notes in Computer Science, vol. 274)*, pages 385–407. Springer-Verlag, 1987.

[74] S.C. Wray and J. Fairbairn. Non-strict languages — programming and implementation. *Computer Journal*, 32(2):142–151, 1989.

Index