Replacing Function Parameters by Global Variables

Peter Sestoft

DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
Electronic mail: sestoft@diku.dk

Speciale. Project Number 88-7-2
October 1988

SUMMARY: This work concerns efficient implementation of strict functional languages or, equivalently, automatic transformation of functional programs into imperative ones. Specifically, we investigate when function parameters can be safely replaced by global variables. This replacement is called *globalization*. The objective of globalization is to reduce the time and space cost of stack (or heap) allocation of function parameters when possible. The tools employed are automatic analysis and transformation of programs. In particular we present an interference analysis that decides whether a given function parameter may safely be globalized, and an algorithm that finds globalizable variable groups: sets of function parameters that may be globalized using only one global variable for each set. The analyses and transformations are based on formal operational semantics to facilitate correctness proofs.

The main contribution of this report is to introduce the concepts of definition-use path, path semantics, interference, and definition-use grammar. A *definition-use path* is a linear recording of the definitions and uses of variables during a computation. The *path semantics* of a language prescribes the path for a computation and is an extension of the operational semantics. *Interference* in a path means that the value of a variable becomes modified (by a redefinition) before its last use. A non-interfering variable is globalizable. A *definition-use grammar* is constructed from a program and the path semantics, and derives a superset of all possible paths for the program. The *interference analysis* is done by an analysis of the grammar, and it safely approximates the "exact" interference in the program. Finally, the globalization transformation uses the results of the interference analysis.

The techniques apply to both first order and higher order languages with strict (*i.e.*, call by value) semantics. We compare our techniques to those used in related work reported in the literature.

Table of Contents

Preface 4
Acknowledgements 4
1. Introduction and Overview 5
1.1 Globalization 5
1.2 Implementation of Strict Functional Languages 6
1.3 Semantics-Directed Compiler Generation 6
1.4 Related Work 7
1.5 Outline of the Report 8
2. Replacing Parameters by Global Variables 10
2.1 Introduction 10
2.2 A First Order Example Language L 13
2.3 Definition-Use Paths and Path Semantics 20
2.4 Interference and Variable Groups 25
3. Definition-Use Grammars: Interference Analysis 33
3.1 Definition-Use Grammars from Path Semantics 33
3.2 Interference Analysis on Grammars 40
3.3 Construction of a Non-Interfering Variable Grouping 46
3.4 The Globalization Transformation for L 51
3.5 Overview: The Globalization Method for L 58
4. Extension to Higher Order Functions 59
4.1 A Higher Order Example Language H 59
4.2 Path Semantics for H 66
4.3 Closure Analysis 68
4.4 Definition-Use Grammar Construction for H 71
4.5 The Globalization Transformation for H 73
4.6 Overview: The Globalization Method for H 74
5. Correctness for H 75
5.1 Correctness of the Closure Analysis 75
5.2 Correctness of the Globalization Transformation for H 77
6. Related Work 82
6.1 D. A. Schmidt: Detecting Global Variables in Denotational Specifications . . 82
6.2 U. Kastens and M. Schmidt: Lifetime Analysis for Procedure Parameters . . 85
6.3 A. Pettorossi: Recursive Programs Which Are Memory Efficient 89
6.4 Overview 90
6.5 Other Work 91
7. Evaluation and Results 92
7.1 Quality of the Globalization Criteria 92
7.2 Expected Cost of the Analyses and Transformations 92
7.3 Benefits to Be Expected from Globalization 93
8. Directions and Open Problems 94
8.1 Improvements of This Work 94
8.2 Other Uses of Path Semantics 96
8.3 Application to Lazy Languages?. . . . 99
9. Conclusion 100
10. Glossary of Symbols 101
11. References 104

Last page 107

PREFACE

This report discusses a particular optimizing transformation applicable to functional languages: the replacement of function parameters by global variables. The report is a "speciale", corresponding to an M.Sc. thesis. It was written at DIKU (the Department of Computer Science at the University of Copenhagen) in the period from June 1987 to September 1988.

Neil D. Jones suggested the subject of this work and was the supervisor of the project. The original goal was to apply David A. Schmidt's work on single-threading in denotational specifications to the area of functional programs in general. It appears that we have succeeded in this, but the technique we have developed is quite different from Schmidt's.

The report should be relevant to people interested in implementation, optimization, analysis, or transformation of eager functional programming languages. The presentation is somewhat formal, with definitions, propositions, and lemmas, but hopefully not unbearably so. We have tried to give examples and intuitive explanations throughout. We have not actually programmed any of the analyses or algorithms presented, but it should be fairly easy to make experimental implementations.

Prior knowledge about structural operational semantics, context free grammars, graphs, and implementation of eager functional programming languages will be beneficial, but is not indispensable. We use the concepts in a rather straightforward manner and try to give some introductory explanation in each case.

ACKNOWLEDGEMENTS

Neil D. Jones was the untiring and patient supervisor of this project, and most of the good ideas originate from him. In addition to Neil, I extend thanks to the following people:

- To Austin Melton, Carsten Gomard, Hans Dybkjær, Olivier Danvy, and Torben Mogensen for reading drafts of the report and providing useful comments;
- to the first floor/North wing group, and in particular Helle, Inger, Claude, and Jan, for making me enjoy going to work;
- to the programming language theory subgroup and its guests (Charles Consel, John Hughes, Keisuke Toyama, Richard Kieburtz and many others) for their interest in my work; and
- to the Carlsberg Foundation for financing this work via a scholarship.

Last but not least, I want to thank Lone for her support in ways too numerous to mention.

1. INTRODUCTION AND OVERVIEW

This section motivates our interest in globalization: the replacement of function parameters by global variables. Furthermore, it lists related work and gives an outline of the report.

1.1 Globalization

Globalization means replacement of function parameters by global variables. This transformation applies to (strict) functional programs and introduces global variables and assignments, so the result is an imperative program.

For example, consider the following function definition:

```
f(x) = x*7
```

Evaluation of a call `f(13)` of `f` will create a new environment which binds `x` to 13; the body `x*7` of `f` is evaluated in this environment; and the environment is discarded. Thus the binding of `x` to 13 is temporary. Now if `x` is replaced by global variable `Z`, then the call of `f` will assign 13 to `Z`, and `Z` will retain this value until altered by a new assignment. If `f` is called recursively, `Z` might be altered too early, and the replacement of `x` by `Z` would not be correct.

The goal of globalization is *efficiency*: imperative programs can be implemented more efficiently than functional ones on existent (von Neumann type) computers. The stack or heap management necessary for functional language implementations is more expensive in terms of run-time and storage consumption than fixed global allocation.

In general it will be incorrect to replace just any function parameter by a global variable: this may alter the meaning of the program. Our goal is to devise an analysis that finds those parameters in a given program which *can* be globalized without changing the meaning of the program. This analysis is *automatic* and *safe*: it will say that a parameter is globalizable only if it definitely is. On the other hand, the analysis is *approximate*: it may fail to identify parameters that *are* globalizable. We also design a globalization transformation that does the globalization on the basis of the analysis.

We emphasize that the analysis and transformation must be correct. We give a formal (operational) semantics for the example languages we work with, and prove the correctness of the globalization method on this basis.

The report introduces several concepts to discuss globalization and to develop the analysis and globalization transformation: definition-use path, interference, path semantics, variable group, definition-use grammar and so on. These concepts are introduced and motivated as they are needed in the development.

But first we will elaborate on the motivations for studying globalization.

1.2 Implementation of Strict Functional Languages

Functional programs are easier to write and read than imperative ones. They have simpler semantics because of the absence of assignment statements and similar constructs. Therefore they are also easier to reason about and to prove correct. Moreover, higher order functions provide a simple and powerful abstraction tool which is not easily introduced into imperative languages. These are some of the well-known arguments in favour of using functional languages [Backus 1978], [Henderson 1980], [Turner 1982].

The argument *against* is inefficiency. Implementations of functional languages on present-day (monoprocessor) computers require more expensive storage management than implementations of imperative languages [Aho, Sethi, Ullman 1986], [Henderson 1980], [Steele 1978]. An "imperative programmer" using Pascal, for example, will know when to use a global variable for storing some data. The "pure functional programmer" has no choice: he must pass his data around via parameters, which is less efficient when running the program.

One solution is to let the programmer write his neat (and provably correct) functional program and then *automatically* replace function parameters by global variables wherever this is admissible. The result is a more efficient and equally correct (but probably less neat) imperative program. This way efficiency is achieved without the programmer having to fight the complexities of assignment.

Thus our globalization method is expected to be useful as a technique for optimization of functional programs, or as a technique to aid in the efficient implementation of functional languages.

Our globalization method works for *strict* (or *eager*) functional languages, such as Standard ML and Scheme. These have "evaluate arguments first" or "call by value" semantics. *Lazy* functional languages such as Hope, Miranda, or Lazy ML require completely different kinds of implementations. Therefore the efficiency problems and the optimization techniques for lazy languages are quite different from those of eager languages. We hope, however, eventually to extend the approach of this work to apply to lazy languages also.

1.3 Semantics-Directed Compiler Generation

Semantics-directed compiler generation is concerned with automatic construction of implementations from formal language definitions. Formal language definitions are usually expressed in some functional language, often a variant of the lambda calculus. (This is not always the case, see for instance [Tofte 1984] and [Jensen, Dam 1985]). Therefore a language definition *may* be *executable*. In that case it can be considered an interpreter, called "int", for the defined language. A program "p" in the defined language now can be executed by running the language definition int on p and p's input data.

The efficiency of executing p depends heavily on the efficiency of the interpreter int. But if an automatic analysis can show that the interpreter itself can be implemented in a particularly efficient way, then it may be feasible to execute the defined language efficiently. Clearly, globalization is relevant, and the work by D. A. Schmidt addresses precisely the problem of detecting global variables in denotational language definitions [Schmidt 1985].

The discussion so far concerns *interpretive* implementation of the defined language. *Compilation* may be done by splitting the interpreter's actions into compile time actions (for example tests on the source program's syntax) and run time actions (for example actions that do work prescribed by the source program). This splitting can be carried out by specializing the interpreter or language definition to the source program [Mosses 1979], that is, by some form of partial evaluation [Futamura 1971], [Ershov 1978]. The splitting may be a source to source transformation of the interpreter so the generated object program is written in the same (functional) language as the interpreter. Again, globalization is a relevant optimization that improves the execution efficiency of such object programs. Thus globalization is relevant for postprocessing of programs generated by the partial evaluator mix, for example [Jones, Sestoft, Søndergaard 1988].

It is undesirable to have to analyse and transform every object program to do globalization. We would want to analyze the language definition (int) once and for all, and then apply only such transformations to it that preserve globalizability (during partial evaluation, for instance). This approach is discussed in [Schmidt 1988]. We shall not discuss this any further.

1.4 Related work

The work by D. A. Schmidt on detecting global variables in denotational semantics definitions was the starting point of these investigations [Schmidt 1985]. A more comprehensive discussion of Schmidt's work is given in Section 6.1.

The work by U. Kastens and M. Schmidt on lifetime analysis for procedure parameters has goals and methods very similar to ours [Kastens, Schmidt 1986]. (Note that D. A. Schmidt ≠ M. Schmidt). However, our work was developed independently to a certain point and puts more emphasis on correctness and formalism than theirs. A detailed comparison is made in Section 6.2.

The related subject "destructive operators" (such as "destructive cons") in applicative languages seems to be an active research field. Early work in this direction is [Pettorossi 1978, 1984] (mentioned in Section 6.3) and [Mycroft 1981].

Our work is certainly related to classical optimization techniques such as *live variable analysis* or *definition-use chaining* etc. in compiler technology [Aho, Sethi, Ullman 1986]. The objective of those techniques is to optimize register allocation or to minimize the number of temporary variables and avoid unnecessary copying of values. The techniques are used to optimize the code generated for expressions. They do not achieve the effect of replacing function parameters by global variables.

The Bliss compiler has particularly sophisticated mechanisms for detecting the lifetime of temporary variables in intermediate code. This gives very good register usage [Wulf *et al.* 1975]. The objective again is to generate high quality code for expressions, and the mechanisms are applied only to one subroutine at a time. The effect is that the mechanisms must be very pessimistic about the way subroutines call each other; they cannot take advantage of a certain call pattern, for example. Our analysis will not do any register optimization, but will be more global, and thus will yield savings not possible with the Bliss approach.

In a functional language, a *tail call* in a function is a call which is the last action in an evaluation of the function's body. It is well known that a directly recursive function, all of whose calls are tail calls, can be evaluated without using a stack [Gill 1965], [Knuth 1972]. This is because there is no need to retain the values of the current local variables (that is, function parameters) during evaluation of a new recursive call. Thus the new values for the parameters may be put in the same place as the old ones. Modern implementations of functional languages recognize tail calls and implement them efficiently as jumps without using a stack [Steele 1978], [Peyton Jones 1987].

Our globalization techniques will allow us to globalize function parameters even in the absence of tail calls and thus is more general than the tail call optimization. Our techniques as presented will detect some tail calls automatically. However, handling tail calls in general may require introduction of temporary variables and this complication is not addressed in our approach.

Work under the heading "recursion elimination" has attempted to extend the tail recursion concept. That is, its goal is to identify certain recursion patterns that could be implemented more cheaply than with the general stack mechanism [Haskell 1975], [Strong 1971], [Walker, Strong 1973]. These recursion schemes appear to be rather restricted, however, and recursion elimination seems to have had little practical impact [Steele 1977].

1.5 Outline of the Report

Sections 2 through 5 constitute the core of this report and should be read in the order they appear. They develop concepts and tools for automatic globalization in first order and higher order languages. Not to attack too many problems at once, Sections 2 and 3 introduce all the important ideas in the context of the simple first order language L; then Section 4 presents the modifications necessary for a higher order language H.

Section 2 introduces the globalization problem and gives some examples. To allow for a precise discussion, it introduces a (strict) first order example language L, giving its syntax and operational semantics. It then defines some concepts central to this work: definition-use path and interference in a definition-use path, and gives a so-called path semantics for L based on the operational semantics for L. Finally, the concept of variable group is defined.

Section 3 develops the construction of a definition-use grammar (from the path semantics) for a given program, and an interference analysis for such grammars. An algorithm that uses this analysis to find interference free variable groupings, and the globalization transformation that replaces function parameters by global variables, are both given. The development in Sections 2 and 3 is complete for the first order case. Section 3.5 summarizes the development so far.

Section 4 contains the extension to the higher order case. It defines the syntax, operational semantics, and path semantics for a (strict) higher order example language H. Then it presents a closure analysis that is needed for the grammar construction for higher order functions. The interference analysis and the construction of a non-interfering variable grouping are identical to those for the first order case.

Section 5 concerns the correctness of the closure analysis and the globalization transformation for H.

Section 6 discusses other work closely related to ours. In particular, D. A. Schmidt's work on single-threading in denotational semantics definitions inspired this study, and U. Kastens and M. Schmidt's work on lifetime analysis for procedure parameters seems to use methods very similar to those employed here.

Section 7 briefly assesses our results and returns to the broader context of functional language implementation to discuss the utility of globalization.

Section 8 discusses open problems and possibilities for applying the tools developed here to related problems, in particular the following one: when may a heap (for storing closures and other dynamically allocated objects) be replaced by a stack? The goal of this kind of transformation is the same as the one discussed above: to save time and space by replacing an expensive storage allocation method (heap) by a less expensive one (stack).

Sections 9, 10, and 11 contain a conclusion, a glossary of symbols, and a list of references.

2. REPLACING PARAMETERS BY GLOBAL VARIABLES

This section falls in two parts. First, globalization is discussed and illustrated by means of examples. Second, we introduce a simple first order example language L and a number of concepts to allow for a more precise treatment of the problem. The syntax, an operational semantics and a path semantics for L are given. The concepts are: the definition-use path for a computation (abstracting the sequence of definitions and uses of a function parameter); interference in a path (expressing that the value of a variable is changed before its last use); and the path semantics for a language (prescribing the definition-use path for evaluation of an expression in a given environment).

Here and in Section 3 we treat the first order language L, and in Section 4 we present extensions and modifications necessary for a higher order language H.

2.1 Introduction

To illustrate the problem of replacing function parameters by global variables, we consider a few example programs written in a simple applicative programming language L with strict application (call by value semantics), left to right evaluation order, and lexical scope. It is presented in Section 2.2 below. We shall assume that L is implemented with a stack of activation records for parameter passing, so that a call of function `f` involves the following actions: argument expressions are evaluated from left to right and their values are pushed onto the parameter passing stack. Then the body expression of `f` is evaluated in a new environment that binds `f`'s variables to the argument values on the stack. After evaluation of the body expression, the argument values are taken off the stack again and control passes back to the point of the call of `f`.

The requirement of left to right evaluation order is not essential but simplifies the discussion considerably. Call by value semantics, on the other hand, is a natural and important prerequisite for this development. Call by name, call by need, or even lazy evaluation requires a quite different kind of implementation and does not combine too well with side effects.

We begin with the particularly simple case of an essentially iterative program.

Example 2.1-1: Powers of two. The program has a single defined function `f` with two parameters `r` and `n`, and an initial expression which is a call `f(1,x)` of the function. For given input `x` which must be a nonnegative integer, the program computes 2^x.

```
def f(r,n) = n=0 →r,
                   f(2*r,n-1)
in  f(1,x)
```

Consider the call from `f` to itself. The value of `r` is not needed after evaluation of `2*r`, so the value of `2*r` need not be pushed onto the parameter passing stack. It may be stored in the location formerly occupied by `r`. Likewise, the value of `n` is not needed after `n-1` is computed, and so `n-1` may be stored in the location formerly occupied by `n`. Thus `r` and `n` can be ***globalized***, that is, replaced by global variables `R` and `N`, and the program transformed into this version using no stack:

```
def f() = N=0 →R,
                    R:=2*R;N:=N-1;f()
in  R:=1;N:=x;f()
```
□

In this case the transformation was particularly simple since the global variables could simply be assigned their new values before the call. However, consider the case where the recursive call had the form `g(n*r,n-1)` instead and assume that for some reason the second parameter `n` is to be globalized while `r` is not. Then it would be wrong to replace the call by an assignment to global variable `N` followed by the call as in `N:=N-1;g(N*r)`. This is because `N` would be assigned a new value too early and so `N*r` would yield the wrong result. We conclude that the order of assignments to the global variables is important and must be the same as that of the argument expressions they replace. (Recall that argument evaluation is left to right.)

The language L will be equipped with a simple device that allows us to put assignments into a parameter list. Writing an assignment `N:=N-1` within brackets `[N:=N-1]` in an argument expression means that the assignment is evaluated for its effect on `N`, but no value is pushed on the (parameter passing) stack. Using this notation, the hypothetical call `g(N*r,N-1)` above should be transformed into `g(N*r, [N:=N-1])`. This expression will first evaluate `N*r` and push the result onto the stack, and then evaluate `N-1` and assign the value to `N`, but push nothing onto the stack. While the bracket notation may look strange at first, it is nevertheless quite natural when seen from the implementation level. The bracket notation also obviates the need for an explicit sequencing operator "`;`" in the language. Although the argument list of the call to `g` has two elements, only one argument is pushed onto stack, and `g` has only one parameter.

Using the new notation and returning to Example 2.1-1, the transformed program will be

```
def f() = N=0 →R,
                    f([R:=2*R],[N:=N-1])
in  f([R:=1],[N:=x])
```

The second example program is an interpreter for a (very) simple imperative language µP. The interpreted language has three kinds of commands: assignment, sequencing, and skip; and three kinds of expressions: constants, variables, and addition. Expressions cannot have side effects.

Example 2.1-2: Interpreter for µP. The interpreter takes as input a source program in µP and evaluates it with an empty initial store. Function `cmd` executes a command `c` in a given store `sc` and returns a new store. Function `exp` evaluates an expression `e` in store `se` and returns the value of `e`.

```
def cmd(c,sc) =
        c= "skip"      → sc,
        c= "z:=e1"     → update(sc,z,exp(e1,sc)),
        c= "c1;c2"     → cmd(c2,cmd(c1,sc)),
        ...
    exp(e,se) =
        e= constant "k" → k,
        e= variable "z" → lookup(z,se),
        e= "e1+e2"     → plus(exp(e1,se),exp(e2,se)),
        ...
in  cmd(source, emptystore)
```

Update, lookup, and plus are basic functions and emptystore is a basic constant. Variable sc in cmd is globalizable, for in neither of the recursive calls from cmd to itself is the old value of sc used after evaluation of the argument expressions. (Here the assumption about call by value is important). Variable se of exp is also globalizable because all its incarnations have the same value in an evaluation of exp. Thus sc and se can be replaced by global variables SC and SE, respectively. Moreover, they can be replaced by a common global variable S, since SE is just a copy of the current SC and exp does not modify SE at all. Notice that the call exp(e,se) in exp would be transformed into exp(e_1, [S:=S]) by a naive replacement of the argument expressions by assignments. Trivial assignments of this form will be left out, however. The resulting transformed program using S for sc and se is:

```
def cmd(c) =
        c= "skip"      → S,
        c= "z:=e1"     → update(S,z,exp(e1)),
        c= "c1;c2"     → cmd(c2, [S:=cmd(c1)]),
        ...
    exp(e) =
        e= constant "k" → k,
        e= variable "z" → lookup(z,S),
        e= "e1+e2"     → plus(exp(e1),exp(e2))
        ...
in  cmd(source, [S:=emptystore])
```
□

This is a quite natural and pleasing result. (Note however that S is still explicitly passed as a parameter to the basic functions update and lookup).

In an imperative language the store is updated in a sequential manner and therefore it is no surprise that it can be represented by one global variable. It is entirely another problem to recognize from an interpreter written in a functional style that the variable(s) representing the store can be globalized. The goal of this work is to do this recognition automatically.

The recursive call in the Example 2.1-1 program above is a *tail call* (or is in tail position). A call in a function body is a tail call if evaluation of the call is the last action in evaluation of the body. It is well known that a tail call can be implemented by a jump to the called function: there is no need to return to the place of the call [Gill 1965], [Knuth 1974]. Also, after evaluation of the argument expressions, the values of local variables in the calling function will no longer be needed and therefore can be discarded *before* the jump. Our techniques for globalization will not incorporate explicit detection of tail calls. The techniques will detect many cases but not all, for in general transformation of a tail call into a jump will require the introduction of temporary variables (or reordering of argument expressions) and this is not attempted. Traditional compiler techniques such as live variable analysis can be used for these purposes, and these analyses need be done only locally in this case [Aho, Sethi, Ullman 1986]. The present approach is easily modified to assume that any temporary variables needed are introduced, and will then detect all tail calls. A few simple changes to the path semantics (Section 2.3.2) and the grammar construction (Section 3.1.2) will do.

The target language for the globalization transformation contains the bracket notation `[...]` not found in usual functional programming languages. To do globalization in the language Scheme, for instance, a call such as `f([R:=2*R],[N:=N-1])` must be replaced by a sequential expression doing the assignments before the call is evaluated:

```
(begin (set! R (* 2 R)) (set! N (- N 1)) (f)).
```

This transformation is simple and can be done locally, but in general it may require introducing temporary variables. This will not be discussed further.

2.2 A First Order Example Language L

This section introduces the first order example language L. The language is introduced informally and its syntax and an operational semantics will be given. The example programs shown above were written in L, so no further examples are given here. We start by describing the applicative part of L, then its imperative extensions.

2.2.1 The Applicative Part of L

The language L is a strict (*i.e.*, call by value) first order language of recursive functions. Here we present the purely applicative parts of the language which will be treated by the interference analysis in later sections.

An L program consists of a finite non-empty set of recursion equations (function definitions), indexed by a finite set I, and an initial expression e^0 to evaluate:

$$\begin{array}{ll} \texttt{def} & \{f^i(x^i_1, \ldots, x^i_{arity(f^i)}) = e^i\}_{i \in I} \\ \texttt{in} & e^0 \end{array}$$

The f^i are referred to as (defined) function symbols and the x^i_1, ... are called the *parameters* or the (local) *variables* of f^i. The name of a variable shows its position: x^i_j denotes parameter position j of function $\texttt{f}^i$. In examples we (have not named and) shall not name variables this way, but we require that all variable names in a program are distinct. This is just to avoid excessively complicated notation. Each equation is called a *function definition*, and the expression in the definition of function $\texttt{f}^i$ is called the *body* of $\texttt{f}^i$. Every defined function `f` has a fixed non-negative arity (*i.e.*, number of parameters) designated by arity(`f`). The language has lexical scope rules for variables and so only the variables $\texttt{x}^i{}_1, \ldots, \texttt{x}^i{}_{\text{arity}(\texttt{f}^i)}$ of function $\texttt{f}^i$ may appear in its body $\texttt{e}^i$, $i \in I$. A program is evaluated in an initial environment that provides values for the variables occurring in the initial expression $\texttt{e}^0$.

Expressions in the language follow the syntax

e	::=	x	- variable (parameter)
	|	$\texttt{A}(\texttt{e}_1, \ldots, \texttt{e}_a)$	- call of basic function `A`
	|	$\texttt{e}_1 \rightarrow \texttt{e}_2, \texttt{e}_3$	- (McCarthy) conditional
	|	$\texttt{f}(\texttt{e}_1, \ldots, \texttt{e}_a)$	- call of defined function `f`

It is assumed that a suitable set of basic functions `A` is given. A parameterless basic function is considered a constant (numeral, boolean, ...), and a call `A()` will usually be written `A`. Evaluation of an expression takes place in an environment which binds variables to their values. A variable occurrence x evaluates to the value x is bound to in the environment. A basic function call $\texttt{A}(\texttt{e}_1, \ldots, \texttt{e}_a)$ is evaluated by first evaluating all its argument expressions from left to right to obtain their values, and then applying the denotation of the basic function to these values. A conditional expression $\texttt{e}_1 \rightarrow \texttt{e}_2, \texttt{e}_3$ is evaluated by evaluating $\texttt{e}_1$ to obtain a boolean value. If it is true, then $\texttt{e}_2$ is evaluated; if it is false, then $\texttt{e}_3$ is. A call $\texttt{f}(\texttt{e}_1, \ldots, \texttt{e}_a)$ is evaluated by evaluating its argument expressions from left to right to obtain their values; then a new local environment is created that binds the variables of the called function `f` to these values; and the body of `f` is evaluated in this new environment.

Notice that the conditional is strict only in its first parameter position whereas all functions, defined as well as basic, are strict in all parameter positions, and the evaluation order is left to right and inside-out. We shall use the typographical convention that L program phrases are written in the `typewriter font`. The set of L expressions is denoted by LExpr.

This completes the description of the syntax and informal semantics of purely applicative L programs.

2.2.2 Operational Semantics

The semantics of L can be described more formally by means of structural operational semantics [Plotkin 1981], [Kahn 1987]. Such a description is a set of inference rules, each involving one or more *evaluation judgements* of the form

$$\rho \vdash e \Rightarrow v$$

where ρ is an environment that binds (local) variables to values, e is an L expression, and v is a value. This judgement reads as follows: in environment ρ, evaluation of expression e will terminate with result v (or may terminate with result v, depending on the language being deterministic or not). This is a "big-step" semantics: the judgement shows the *final* value of evaluation of e. The notation with "$\vdash$" (which may be pronounced "entails") indicates the dependency of e's evaluation on the current environment ρ. In general the evaluation depends also on the rest of the program (namely, in case e contains a call of a defined function), so we really ought to write "pgm, $\rho \vdash e \Rightarrow v$" (where pgm is the program) to make this dependency explicit. However, the program remains constant throughout evaluation so we shall just assume that a program pgm is available and not burden the evaluation judgements with this extra symbol.

A judgement may be understood as a proposition about the value of an expression in a given environment. Then evaluation (program execution) is theorem proving, and the semantics of a language can be described using semantics rules which are in essence logical inference rules involving (*i.e.*, relating) one or more evaluation judgements. We take the rules for evaluation of a conditional expression $e_1 \rightarrow e_2, e_3$ to illustrate this approach:

$$\frac{\begin{array}{l}\rho \vdash e_1 \Rightarrow \text{true} \\ \rho \vdash e_2 \Rightarrow v\end{array}}{\rho \vdash e_1 \rightarrow e_2, e_3 \Rightarrow v}$$

$$\frac{\begin{array}{l}\rho \vdash e_1 \Rightarrow \text{false} \\ \rho \vdash e_3 \Rightarrow v\end{array}}{\rho \vdash e_1 \rightarrow e_2, e_3 \Rightarrow v}$$

The rules state that if in the given environment ρ, e_1 may evaluate to true and e_2 may evaluate to v, then the entire conditional expression may evaluate to v; and that if e_1 may evaluate to false and e_3 may evaluate to v, then the entire conditional expression may evaluate to v. According to these rules, the conditional is strict in its first position: e_1 *must* terminate with a result for the conditional to have a value. Also, if e_1 evaluates to anything but true or false, then the conditional is undefined (for the above rules are the only ones defining the conditional). Finally, the conditional is non-strict in its second (and third) positions because evaluation of e_2 (or e_3) may fail to terminate with a result and still the conditional may evaluate to a value.

The judgements over the line are called *premises* and the one below is called the *conclusion*. If a rule has no premises, then the line will be omitted. Further conditions on the applicability of a rule may be stated to the right of the rule with the obvious meaning.

The complete set of semantic rules is given below. We assume (cf. above) that a program

$$\text{pgm} = \ \texttt{def}\ \{f^i(x^i_1, \ldots, x^i_{\text{arity}(f^i)}) = e^i\}_{i \in I}\ \texttt{in}\ \ e^0$$

is given along with an initial environment ρ_0 which binds the variables of e^0 to initial values: the input to pgm.

Letting FreeVars(e) be the sets of variables that occur in expression e, and $A \dashrightarrow B$ be the set of partial functions from A to B, we use the following sets and mappings in the semantics rules below:

v:	$\text{Value} = \text{Boolean} \cup \text{Integer} \cup \ldots$	value,
	$\text{Input} = \text{FreeVars}(e^0)$	input parameters of the program,
	$\text{Var}_i = \{x^i_1, \ldots, x^i_{\text{arity}(f^i)}\}$	local variables of function f^i,
x:	$\text{Var} = \bigcup \{ \text{Var}_i \mid i \in I \}$	local variable of defined function,
ρ:	$\text{Env} = \text{Var} \dashrightarrow \text{Value}$	local environment.

The set Input of input parameters must be disjoint from the set Var of local variables. For any given program, Var is a finite set and so ρ is a finite function. We write

$\varnothing$	for the empty function (everywhere undefined),
$\rho[x \mapsto v]$	for $\lambda z.\text{if } z=x \text{ then } v \text{ else } \rho(z)$,
$\rho[x_1 \mapsto v_1, x_2 \mapsto v_2]$	for $\rho[x_1 \mapsto v_1][x_2 \mapsto v_2]$, and
$[x \mapsto v]$	for $\varnothing[x \mapsto v]$.

The result of evaluating the entire program is the result of evaluating the initial expression in some initial environment ρ_0: Input $\rightarrow$ Values.

$$\text{(P1)} \quad \frac{\rho_0 \vdash e^0 \Rightarrow v}{\rho_0 \vdash \texttt{def}\ \{f^i(x^i_1, \ldots, x^i_{\text{arity}(f^i)}) = e^i\}_{i \in I}\ \texttt{in}\ \ e^0 \Rightarrow v}$$

The judgement below the line (that is, in the conclusion) concerns evaluation of programs, not expressions. To be completely formal we should subscript the "$\vdash$" symbol with "program" to make this difference explicit: $\vdash_{\text{program}}$, but we will not do that. The rules for evaluation of expressions are:

(E1) $\rho \vdash \mathtt{x} \Rightarrow \mathrm{v}$ where $\rho(\mathtt{x}) = \mathrm{v}$

(E2) $$\frac{\rho \vdash \mathtt{e}_j \Rightarrow \mathrm{v}_j \text{ for } j=1,\ldots,a}{\rho \vdash \mathtt{A}(\mathtt{e}_1, \ldots, \mathtt{e}_a) \Rightarrow \mathrm{v}}$$ where $a = \text{arity}(\mathtt{A})$ and $\mathrm{v} = \mathtt{A}(\mathrm{v}_1, \ldots, \mathrm{v}_a)$

(E3) $$\frac{\begin{array}{l}\rho \vdash \mathtt{e}_1 \Rightarrow \text{true}\\ \rho \vdash \mathtt{e}_2 \Rightarrow \mathrm{v}\end{array}}{\rho \vdash \mathtt{e}_1 \rightarrow \mathtt{e}_2, \mathtt{e}_3 \Rightarrow \mathrm{v}}$$

(E4) $$\frac{\begin{array}{l}\rho \vdash \mathtt{e}_1 \Rightarrow \text{false}\\ \rho \vdash \mathtt{e}_3 \Rightarrow \mathrm{v}\end{array}}{\rho \vdash \mathtt{e}_1 \rightarrow \mathtt{e}_2, \mathtt{e}_3 \Rightarrow \mathrm{v}}$$

(E5) $$\frac{\begin{array}{l}\rho \vdash \mathtt{e}_j \Rightarrow \mathrm{v}_j \text{ for } j=1,\ldots,a\\ \rho' \vdash \mathtt{e}^i \Rightarrow \mathrm{v}\end{array}}{\rho \vdash \mathtt{f}^i(\mathtt{e}_1, \ldots, \mathtt{e}_a) \Rightarrow \mathrm{v}}$$ where $a = \text{arity}(\mathtt{f}^i)$, $\mathtt{e}^i$ is the body of function $\mathtt{f}^i$, and $\rho' = [\mathtt{x}^i_1 \mapsto \mathrm{v}_1, \ldots, \mathtt{x}^i_a \mapsto \mathrm{v}_a]$

Comparing these rules with the informal description of the semantics given in Section 2.2.1 above should show that they are straightforward translations into symbols of the informal description.

Evaluation of an expression e in environment ρ using semantic rules may be seen as the process of building a finite *evaluation tree* whose root is a judgement of form $\rho \vdash \mathtt{e} \Rightarrow \mathrm{v}$. Each branching point is an instance of a semantic rule: its stem is the conclusion of the rule instance and its branches are the premises of the rule instance. The leaves of the evaluation tree are instances of rules that have no premises (here: evaluation of variable occurrences by rule E1 or of parameterless basic function calls by rule E2). The evaluation tree is a Gentzen-style (natural deduction) proof tree for the proposition that e evaluates to v in environment ρ [Lyndon 1966]. It is easy to see that L is deterministic: if $\rho \vdash \mathtt{e} \Rightarrow \mathrm{v}$ and $\rho \vdash \mathtt{e} \Rightarrow \mathrm{v}'$, then $\mathrm{v} = \mathrm{v}'$.

Example 2.2.2-1: Evaluation tree for evaluation of the expression "x=y → 21, z" in environment $\rho = [\mathtt{x}\mapsto 7, \mathtt{y}\mapsto 9, \mathtt{z}\mapsto 13]$. The infix basic function "=" has arity two. The evaluation tree involves one application of rule E4, one of rule E2, and three applications of rule E1:

$$\frac{\dfrac{\rho \vdash \mathtt{x} \Rightarrow 7 \quad \rho \vdash \mathtt{y} \Rightarrow 9}{\rho \vdash \mathtt{x=y} \Rightarrow \text{false}} \qquad \rho \vdash \mathtt{z} \Rightarrow 13}{\rho \vdash \mathtt{x=y}\rightarrow\mathtt{21,z} \Rightarrow 13}$$ □

The operational semantics does not explicitly deal with nonterminating evaluations: these would correspond to infinite evaluation trees. In a denotational semantics, a nonterminating evaluation of expression e in environment ρ would yield a special "value" ($\perp$, say) while in the operational semantics there would just be no (ordinary) value v such that $\rho \vdash \mathtt{e} \Rightarrow \mathrm{v}$ according to the semantic rules. This has the implication that our correctness arguments (in later sections) apply only to terminating evaluations. Henceforth, by evaluation we mean terminating evaluation.

2.2.3 The Imperative Part of L

Later on, we shall transform applicative L programs to use global variables, so we must extend L with imperative constructs. These are given here but they are not used before Section 3.4 and so may be skipped until then. (I admit that the imperative extensions could be more elegant. See Section 8.1.1).

The language is enriched with the concepts of global variable and global state (to hold the values of the global variables). The syntactic category of expressions is extended as follows:

```
e    ::=  X              - reference to global variable X
       |  X:=e           - assignment of e's value to X
       |  [X:=e]         - evaluate X:=e; do not push to stack
```

An occurrence of a global variable X evaluates to its value in the current global state. Evaluation of X:=e is done by evaluating e to obtain a value and then modifying the global state at X to have this value. The value of X:=e is the value of e. An expression of form [X:=e] can occur only as argument expression in a call to a defined function and is evaluated only for its effect on the global state: X:=e is evaluated but its value does not become bound in the new environment built for evaluation of the called function's body. Global variables are written in upper case to distinguish them from the others, and we require that the set of global variables is disjoint from the set of local variables. Evaluation of a basic function A can neither access nor modify the state of global variables. Evaluation of a program starts in an initial state σ_{init} that contains no bindings at all, and so it is illegal to reference a global variable before it has been assigned a value.

The arity a = arity(f^i) of f^i is its number of local variables $x^i{}_1,\ldots, x^i{}_a$. The number m of argument expressions in a call $f^i(e_1,\ldots,e_m)$ to f^i may vary, but the number of argument expressions e_j which do not have form [...] must equal arity(f^i). The set of expressions in imperative L is denoted by LIExpr.

The evaluation of an expression now also depends on the global state σ, so the evaluation judgements are modified to include the dependency on the global state and to show the final ("output") state of evaluation:

$$\rho, \sigma_1 \vdash e \Rightarrow v, \sigma_2.$$

This reads: in environment ρ and global state σ_1, evaluation of expression e may terminate in state σ_2 with result v. All the rules of the operational semantics must be changed to reflect the presence of a global state which can be updated at any time. The modifications required are straightforward:

they must reflect the left-to-right order of subexpression evaluation. In addition, three new rules must be added to define the new expressions.

The sets and mappings given above must be extended with a finite set GloVar of global variables, disjoint from the set Var of local variables and from Input, the set of input parameters.

$\mathtt{X}$: GloVar = $\{\mathtt{X}_1, \ldots\}$	global variable,
σ: State = GloVar $\dashrightarrow$ Value	global state,
$\sigma_{init} = \varnothing$: State	the empty initial state.

Because GloVar is finite, σ is a finite function. Now the are three kinds of "variables": local variables $x \in$ Var, global variables $\mathtt{X} \in$ GloVar, and input parameters $x \in$ Input. The sets Var, GloVar, and Input must be mutually disjoint. The program and expression evaluation rules are:

(IP1)
$$\frac{\rho_0, \sigma_{init} \vdash e^0 \Rightarrow v, \sigma}{\rho_0 \vdash \mathtt{def}\ \{\mathtt{f}^i(x^i_1, \ldots, x^i_{arity(\mathtt{f}^i)}) = e^i\}_{i\in I}\ \mathtt{in}\ e^0 \Rightarrow v}$$

(IE1) $\rho, \sigma \vdash x \Rightarrow v, \sigma$ where $x \in$ Var and $\rho(x) = v$

(IE2)
$$\frac{\rho, \sigma_{j-1} \vdash e_j \Rightarrow v_j, \sigma_j \quad \text{for } j=1,\ldots,a}{\rho, \sigma_0 \vdash \mathtt{A}(e_1, \ldots, e_a) \Rightarrow v, \sigma_a}$$
where $a = arity(\mathtt{A})$
and $v = \mathtt{A}(v_1, \ldots, v_a)$

(IE3)
$$\frac{\begin{array}{l}\rho, \sigma_0 \vdash e_1 \Rightarrow true, \sigma_1\\ \rho, \sigma_1 \vdash e_2 \Rightarrow v, \sigma_2\end{array}}{\rho, \sigma_0 \vdash e_1 \rightarrow e_2, e_3 \Rightarrow v, \sigma_2}$$

(IE4)
$$\frac{\begin{array}{l}\rho, \sigma_0 \vdash e_1 \Rightarrow false, \sigma_1\\ \rho, \sigma_1 \vdash e_3 \Rightarrow v, \sigma_2\end{array}}{\rho, \sigma_0 \vdash e_1 \rightarrow e_2, e_3 \Rightarrow v, \sigma_2}$$

(IE5)
$$\frac{\begin{array}{l}\rho, \sigma_{j-1} \vdash e_j \Rightarrow v_j, \sigma_j \quad \text{for } j=1,\ldots,m\\ \rho', \sigma_m \vdash e^i \Rightarrow v, \sigma\end{array}}{\rho, \sigma_0 \vdash \mathtt{f}^i(e_1, \ldots, e_m) \Rightarrow v, \sigma}$$
where $m \geq a = arity(\mathtt{f}^i)$,
e^i is the body of function $\mathtt{f}^i$,
and $\rho' = [x^i_j \mapsto v_{h(j)}]$ for $j=1,\ldots,a$,
where $e_{h(1)},\ldots,e_{h(a)}$ all $\neq$ [...]

(IE6) $\rho, \sigma \vdash \mathtt{X} \Rightarrow v, \sigma$ where $\mathtt{X} \in$ GloVar and $\sigma(\mathtt{X}) = v$

(IE7)
$$\frac{\rho, \sigma_0 \vdash e \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \mathtt{X{:=}e} \Rightarrow v, \sigma_2}$$
where $\sigma_2 = \sigma_1[\mathtt{X} \mapsto v]$

(IE8)
$$\frac{\rho, \sigma_0 \vdash e \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \mathtt{[e]} \Rightarrow v, \sigma_1}$$

In rule IE5, the function $h:\{1,..,a\}\rightarrow\{1,..,m\}$ must be strictly increasing so that $e_{h(1)},...,e_{h(a)}$ is a subsequence of $e_1, ..., e_m$. The language is still deterministic: if $\rho, \sigma_0 \vdash e \Rightarrow v, \sigma_1$ and $\rho, \sigma_0 \vdash e \Rightarrow v', \sigma'_1$, then $v = v'$ and $\sigma_1 = \sigma'_1$. This semantics is an extension of that for applicative L: on applicative L programs they yield the same result.

In this section we defined first the applicative and then the imperative parts of the language L. For now only the applicative part is studied. Throughout the report we take *variable* to mean *local variable*, designating a parameter position. Global variables will always be explicitly referred to as such.

2.3 Definition-Use Paths and Path Semantics

In this section we shall first develop a tool to describe the order in which (local) variables are defined and used during evaluation of an L program. This tool is called a *definition-use path* (*du-path* for short) and is a kind of structured execution trace: a linear recording of the actions taken during evaluation. This idea is presented in Section 2.3.1.

In Section 2.3.2 we shall show how to extend the standard operational semantics for L to a *path semantics* that prescribes the definition-use path for evaluation of a given program in a given initial environment.

2.3.1 Definition-Use Paths

A definition-use path (du-path) is a finite sequence of the actions done during an evaluation. The actions of interest to us are those that concern definition and use of local variables (function parameters):

- Use of variable x: when is x referenced during evaluation. Symbol: $\uparrow$x
- Definition of variable x: when does variable x become defined (or redefined). That is, when is a (new) value for x pushed on the parameter passing stack. Symbol: $\downarrow$x
- Definition of variable x by copying: when does x become defined by a copying of another variable's value. That is, when is the value of some variable y pushed onto the parameter stack to become the new value of x. Symbol: y»x

Notice the symbols for the three kinds of actions. Mnemonically, $\uparrow$x = $\underline{u}$p x = $\underline{u}$se x; $\downarrow$x = $\underline{d}$own x = $\underline{d}$efine x; and y»x = copy y to x. The roles of the use and definition actions should be quite clear, whereas that of the copy action deserves some explanation. The intention is that a copy action y»x is a form of potential definition. If x and y are allocated as one common global variable (or they are the same local variable), then the copy action is not a definition at all, since it cannot change the value of x. But if x and y cannot be allocated as a common global variable (and are not the same local variable), it *is* a definition that may change the value of x. The use of this will become clearer in Section 2.4.2 on variable groups.

The du-paths are to be used in the interference criteria below. Their purpose is to detect the archetypical situation that prevents a variable from being allocated globally: a (re)definition interferes between the definition of a variable and its last use, which will then return the wrong value. To detect interference correctly, we must know which definition is the right one corresponding to a given use. In the simple first order language L, this is determined by the nesting of function calls during evaluation. Therefore the du-paths will not merely be linear sequences of actions, but will have a structure corresponding to call nesting. In a du-path, evaluation of a function call `f(e`$_1$`,...,e`$_a$`)` is represented by a ***function call structure*** which is a balanced bracket structure with two subpaths $\pi_1\delta_1 \ldots \pi_a\delta_a$ and π like this:

$$< \pi_1\delta_1 \ldots \pi_a\delta_a \lozenge \pi >.$$

The first subpath $\pi_1\delta_1 \ldots \pi_a\delta_a$ corresponds to the call prelude. Path π_j describes the evaluation of argument expression e_j, and δ_j describes the definition of `f`'s parameter x_j, *i.e.*, the pushing of its value onto the stack. The second subpath π corresponds to evaluation of `f`'s body. Thus operationally "<" marks the beginning of the call prelude, "◊" marks the passing of control from the call point to the function body (and hence to a new environment), and ">" marks the passing back of control (to the old environment). Notice that the definition and copy symbols (such as δ_j) can appear only in call preludes $\pi_1\delta_1 \ldots \pi_a\delta_a$ because they denote the action of pushing values onto the parameter passing stack, and that both the π_j and π may contain nested function call structures.

The language of du-paths is the set Path of paths built from uses ($\uparrow$x) and function call structures (which may involve $\downarrow$x and y»x), closed under formation of finite sequences and function call structures. Formally, for a given set Var of (local) variables, the alphabets of primitive symbols are

Use	$= \{ \uparrow x \mid x \in \text{Var} \}$,	- use symbols
Copy	$= \{ y \gg x \mid x, y \in \text{Var} \}$,	- copy symbols
Def	$= \{ \downarrow x \mid x \in \text{Var} \} \cup \text{Copy}$,	- definition and copy symbols
Σ	$= \{ <, >, \lozenge \} \cup \text{Use} \cup \text{Def}$,	- all symbols

and the set Path of paths is defined inductively as the smallest set satisfying

$$\text{Path} \quad = \text{Use} \cup \text{Path}^* \cup \{ <\pi_1\delta_1 \ldots \pi_a\delta_a \lozenge \pi> \mid \pi, \pi_j \in \text{Path};\ \delta_j \in \text{Def for } j=1,\ldots,a \}.$$

We denote the concatenation of two paths π_1 and π_2 by juxtaposition $\pi_1\pi_2$, and do not distinguish the symbol $\uparrow$x, say, from the path of length one containing just this symbol. The empty path (of length zero) is written ε. To grasp the idea, consider some du-paths for evaluation of the example programs from Section 2.1.

Example 2.3.1-1: A du-path for an evaluation of the powers of two algorithm (Example 2.1-1). Evaluating this program on input x=1, first the initial call `f (1, 1)` is evaluated, yielding $<\downarrow r \downarrow n \ \Diamond$ ...> for a start. Then the body of `f` is evaluated: first `n=0` is evaluated, implying a use $\uparrow n$ of `n`. But n=1≠0, so the second branch of the conditional is evaluated. This is a call to `f` and so yields a new call structure $<\pi_1 \Diamond \pi_2>$. The prelude π_1 is $\uparrow r \downarrow r \uparrow n \downarrow n$ because setting up the parameters comprises a use of `r` to compute `2*r`; a push of this value onto the stack thus defining `r`; a use of `n` to compute `n-1`; and a push of this value thus defining `n`. The path π_2 corresponding to evaluation of the body is $\uparrow n \uparrow r$: first the test `n=0` requires a use of `n`, then (because n=0 in fact), the first branch of the conditional is evaluated, which is just a use of `r`. The resulting path is

$<\downarrow r \downarrow n \ \Diamond$
  $\uparrow n < \uparrow r \downarrow r \uparrow n \downarrow n \ \Diamond$
    $\uparrow n \uparrow r >>$ □

Example 2.3.1-2: A du-path for an evaluation of the μP interpreter (Example 2.1-2). Evaluation of the interpreter on program `source` = "a:=1+3" yields the path

$<\downarrow c \downarrow sc \ \Diamond$ (1)
  $\uparrow c \uparrow c \uparrow sc \uparrow c <\uparrow c \downarrow e \uparrow sc \ sc \gg se \ \Diamond$ (2)
    $\uparrow e \uparrow e \uparrow e <\uparrow e \downarrow e \uparrow se \ se \gg se \ \Diamond$ (3)
      $\uparrow e \uparrow e >$ (4)
    $< \uparrow e \downarrow e \uparrow se \ se \gg se \ \Diamond$ (5)
      $\uparrow e \uparrow e >>>$ (6)

Briefly, in line 1 `c` and `sc` become defined in the prelude of the initial call to `cmd`. In line 2 the body of `cmd` is evaluated: first the tests c="skip" and c="z:=e_1" are evaluated, `sc` is used and `c` is used (to get `z` which is a part of `c`); then, in the prelude to the call of `exp`, `c` is used (to get e_1 which is a part of `c`), `e` becomes defined, `sc` is used and copied to `se`. Line 3: in the body of `exp`, `e` is tested (and hence used) three times and in the prelude to the first recursive call to `exp`, `e` is used (to get e_1), and its value is pushed to define `e`; finally `se` is used and copied to `se`. Line 4: in the body of this recursive call to `exp`, `e` is tested (and hence used), then used again (to get `k`), and the call returns (yielding a ">" in the path). Then the completely similar second recursive call to exp is evaluated: the prelude is in line 5 and the body in line 6. □

Example 2.3.1-3: Another du-path for an evaluation of the μP interpreter (Example 2.1-2). Evaluation of the interpreter on program `source` = "skip;skip" yields the path

$<\downarrow c \downarrow sc \ \Diamond$ (1)
  $\uparrow c \uparrow c \uparrow c <\uparrow c \downarrow c <\uparrow c \downarrow c \uparrow sc \ sc \gg sc \ \Diamond$ (2)
    $\uparrow c \uparrow sc>$ (3)
  $\downarrow sc \ \Diamond$ (4)
    $\uparrow c \uparrow sc>>$ (5)

In line 1, `c` and `sc` become defined in the initial call to `cmd`. In line 2, `c` is tested (and hence used) three times; the prelude of the outer recursive call to cmd begins with a use of c (to get c_2, a part of c), and a push of this value; then the prelude of the inner recursive call to cmd begins: use of c, definition of c, use of sc, and copy of sc. In line 3, evaluation of the body requires a test of c and one use of sc. In line 4, the value computed by the inner recursive call is pushed to define sc, and in line 5 the body of cmd is evaluated once more: test of c, use of sc. □

After these examples of paths, let us define the notion of use level in a path. Consider the path (shown in Example 2.3.1-1) corresponding to evaluation of f(1,1):

```
<↓r  ↓n ◊ ↑n < ↑r↓r↑n ↓n ◊↑n↑r >>
     #1   #2          #3 #4   #5
```

Clearly there are two different levels of use of variable n, say. The uses of n marked #2 and #3 are level 1 uses, corresponding to the first level of nesting of the recursive function calls, and refer to the definition of n at #1. (At #1, in the prelude of the first call to f, a new value of n is pushed onto the parameter passing stack and at #2 and #3, in the body of f, the value is used). The third use of n (at #5) is a level 2 use, corresponding to the new definition of n at #4.

In general, the use level increases by 1 when control passes from a call point to the called function's body, and decreases by 1 when control passes back. In terms of du-paths, the use level increases by 1 after a "◊" and decreases by 1 after a ">". We define the initial level of uses in any path to be 0.

2.3.2 Path Semantics for L

We saw in the preceding section that there corresponds a du-path to every (terminating) evaluation of an L program.

Here we shall show how to extend the standard operational semantics from Section 2.2.2 into a *path semantics* that determines the du-path for a given L program and a given initial environment. We shall give this only for the applicative part of L. The path semantics properly extends the operational semantics: it gives the final result computed by the evaluation and in addition the du-path for the evaluation. The path semantics may thus be seen as a more precise specification or description of the language: whereas the standard semantics specifies which result an evaluation must yield, the path semantics also specifies the exact sequence of the actions done during the evaluation. For example, the standard semantics for (applicative) L does not describe the order of evaluation of argument expressions in a function call; the path semantics does.

The path semantics we give below is completely deterministic in the sense that for a given expression and a given environment, there is (at most) one possible du-path. This is not to say that a "nondeterministic" path semantics cound not be given that would realize one out of several possible evaluations, each with a different du-path, when given an expression and an environment. We shall not study this possibility here, but it would be very useful for the treatment of languages (such as Pascal or Scheme) for which evaluation order of argument expressions is undefined.

Because the path semantics gives more information about an evaluation than the standard semantics does, it cannot be derived mechanically from the latter. On the other hand, the path semantics must reflect the semantics of imperative L to some extent, for it will be used to decide whether local variables can be replaced by global ones. Like the standard operational semantics, the path semantics cannot be used to describe non-terminating computations.

The L path semantics is completely similar in structure to the standard semantics but uses another kind of judgement. It still depends on the environment ρ, but gives two "results" for the evaluation of an expression: the ordinary result value v and the du-path π traced during evaluation of the expression. The standard form of the path semantics judgement is $\rho \vdash e \Rightarrow v, \pi$ which reads: in environment ρ, evaluation of expression e may (or will) trace the path π of actions and yield final result v.

As before, we shall not make the dependency on the actual program explicit in the rules of the semantics; we just assume that an applicative L program pgm is given along with an initial environment ρ_0: Input $\rightarrow$ Values. Again, the result of evaluating the entire program is the result of evaluating the initial expression in the initial environment ρ_0:

$$\text{(PP1)} \quad \frac{\rho_0 \vdash e^0 \Rightarrow v, \pi}{\rho_0 \vdash \texttt{def} \ \{\texttt{f}^i(x^i_1, \ldots, x^i_{\text{arity}(\texttt{f}^i)}) = e^i\}_{i \in I} \ \texttt{in} \ e^0 \Rightarrow v, \pi}$$

The path semantics rules for expressions are all very simple except for the function call rule:

$$\text{(PE1)} \quad \rho \vdash x \Rightarrow v, \uparrow x \qquad \text{where } \rho(x) = v$$

$$\text{(PE2)} \quad \frac{\rho \vdash e_j \Rightarrow v_j, \pi_j \ \text{ for } j=1,\ldots,a}{\rho \vdash \mathrm{A}(e_1, \ldots, e_a) \Rightarrow v, \pi_1 \ldots \pi_a} \qquad \begin{array}{l}\text{where } a = \text{arity}(\mathrm{A}) \\ \text{and } v = \mathrm{A}(v_1, \ldots, v_a)\end{array}$$

$$\text{(PE3)} \quad \frac{\begin{array}{l}\rho \vdash e_1 \Rightarrow \text{true}, \pi_1 \\ \rho \vdash e_2 \Rightarrow v, \pi_2\end{array}}{\rho \vdash e_1 \rightarrow e_2, e_3 \Rightarrow v, \pi_1\pi_2}$$

$$\text{(PE4)} \quad \frac{\begin{array}{l}\rho \vdash e_1 \Rightarrow \text{false}, \pi_1 \\ \rho \vdash e_3 \Rightarrow v, \pi_3\end{array}}{\rho \vdash e_1 \rightarrow e_2, e_3 \Rightarrow v, \pi_1\pi_3}$$

$$\text{(PE5)} \quad \frac{\begin{array}{l}\rho \vdash e_j \Rightarrow v_j, \pi_j \ \text{ for } j=1,\ldots,a \\ \rho' \vdash e^i \Rightarrow v, \pi\end{array}}{\rho \vdash \texttt{f}^i(e_1,\ldots,e_a) \Rightarrow v, <\pi_1\delta_1\ldots\pi_a\delta_a \lozenge \pi>} \qquad \begin{array}{l}\text{where } a = \text{arity}(\texttt{f}^i), \\ e^i \text{ is the body of function } \texttt{f}^i, \\ \rho' = [x^i_1 \mapsto v_1, \ldots, x^i_a \mapsto v_a], \\ \text{and } \delta_j = \Delta(x^i_j, e_j) \text{ for } j=1,\ldots,a\end{array}$$

An auxiliary function Δ: Var $\times$ LExpr $\rightarrow$ Path is used to simulate a copy or a definition:

$$\begin{array}{rcll}\Delta(x, e) & = & y \gg x & \text{if } e \text{ is a variable } y \\ & = & \downarrow x & \text{if } e \text{ is not a variable}\end{array}$$

Like the operational semantics in Section 2.2.2, this path semantics is intended to be a direct translation into symbols of the intended semantics of L programs (in particular their order of evaluation). This path semantics is deterministic, for the evaluation order of L is fixed: left to right. By modifying rules PE2 and PE5 we could make the evaluation right to left. By adding several versions of PE2 and PE5 the path semantics could be made nondeterministic, corresponding to an unspecified order of evaluation in L. In any case, the path semantics must reflect the (possible) order(s) of evaluation in imperative L.

Using the path semantics we can now define the set of possible paths for evaluation of an expression in any given environment ρ. These definitions are needed in Section 3 below.

Definition 2.3.2-1: The set $\Pi(e)$ of paths of expression e is the set of all du-paths that evaluation of e can produce: $\Pi(e) = \{\ \pi \in \text{Path}\ |\ \exists\rho,v.\ \ \rho \vdash e \Rightarrow v, \pi\ \}$. □

Definition 2.3.2-2: The set $\Pi(\text{pgm})$ of paths of program pgm is the set of all du-paths that evaluation of pgm can produce: $\Pi(\text{pgm}) = \{\ \pi \in \text{Path}\ |\ \exists\rho_0,v.\ \ \rho_0 \vdash \text{pgm} \Rightarrow v, \pi\ \}$. □

2.4 Interference and Variable Groups

First, Section 2.4.1 introduces the concept of *interference* in (the du-path of) an evaluation. Interference means that the value of a variable is changed before its last use (by an interfering definition of the same variable). This notion is made precise in terms of the du-path for the evaluation. The importance of the concept is that in case there is no interference with respect to variable x in the du-path π for some evaluation, then x may be allocated as a global variable without changing the the result of the evaluation. Interference and the criteria for its presence in a du-path are central to the theme of this report and will be used heavily in later sections.

Then in Section 2.4.2 the interference concept is refined to apply to *variable groups*. A variable group is a set of variables that are considered for replacement by one and the same global variable.

2.4.1 Interference

Consider an evaluation yielding a du-path π. Path π has *interference* with respect to variable x if replacing x by a global variable X could change the result of the evaluation. This happens when the value of X becomes modified before the last use of that value, by a (re)definition of X. A (re)definition of a global variable will destroy its former value, whereas a (re)definition of a (local) variable will not destroy the old value which is retained on the parameter passing stack. Replacement of x by X may or may not cause an old value to be lost while it is still in use, depending on the evaluation, that is, depending on the du-path for the evaluation.

Let us investigate the various ways a variable x can be interfering in path π and then derive safe criteria for detection of interference. Interference detection is done assuming that x is replaced by a global variable X, so that all uses of x are replaced by references to X, and all definitions of x (that is, pushes of a new value for x onto the stack) are replaced by assignments to X.

We consider the three basic forms of paths in turn (from the inductive definition of the set Path in Section 2.3.1). The value of x at the beginning of the path is accessible only by level 0 uses of x, but may be modified (destroyed) by any redefinition of x preceding a level 0 use. So x is interfering in π if a definition of x in π (at any level) precedes a level 0 use of x; or if x is interfering in any nested call structure in π. Uses at other levels do not come into consideration immediately because they do not refer to the level 0 value of x: the value at the beginning of the path. By definition, uses at higher levels follow a $\Diamond$ in a call structure $< \dots \Diamond \dots >$, and refer to values defined in the prelude of this call structure. Interference in nested call structures is discovered by recursive application of the criteria below.

- The path of length one, $\pi = \uparrow x$: Here x obviously is not interfering.
- Path $\pi = \pi_1 \dots \pi_n$: Here, x is interfering if x is redefined in some π_i and then subsequently used (at level 0) in π_j with i<j. In addition, x may be interfering in one of the π_i.
- Path $\pi = <\pi_1\delta_1 \dots \pi_a\delta_a \Diamond \pi'>$: There are three possibilities. First, x may be interfering in the prelude if x is redefined in $\pi_i\delta_i$ and then subsequently used (at level 0) in π_j with i<j. Secondly, a new value for x may be defined (intentionally) by $\delta_i = \downarrow x$ or $\delta_i = z \gg x$ for some z and then spoiled by a redefinition of x in π_j with i<j before its use (at level 1) in the body π'. Thirdly, x may be interfering in one of the π_i or in the body π'.

We shall define the interference of a path π to be the set of variables interfering in it. To do this formally we first introduce functions for computing the set $U_0(\pi)$ of variables with level 0 uses and the set $D(\pi)$ of variables with definitions in a given path π. For the purpose of this we take a copy $y \gg x$ to be a definition $\downarrow x$ of x unless y and x are identical.

Definition 2.4.1-1: The set $U_0(\pi)$ of variables with a level 0 use and the set $D(\pi)$ of variables with a definition in path π are defined inductively as follows, where $\pi_j \in$ Path, $\delta_j \in$ Def:

$$
\begin{array}{ll}
U_0\text{: Path} \rightarrow \wp(\text{Var}) & \\
U_0(\uparrow x) & = \{ x \} \\
U_0(\pi_1 \dots \pi_n) & = U_0(\pi_1) \cup \dots \cup U_0(\pi_n) \\
U_0(<\pi_1\delta_1 \dots \pi_a\delta_a \Diamond \pi>) & = U_0(\pi_1) \cup \dots \cup U_0(\pi_a)
\end{array}
$$

$$
\begin{aligned}
&D:\ (\mathrm{Path} \cup \mathrm{Def}) \rightarrow \wp(\mathrm{Var}) \\
&D(\uparrow x) &&= \{\} \\
&D(\downarrow x) &&= \{x\} \\
&D(y \gg x) &&= \{x \mid y \text{ and } x \text{ are distinct variables}\} \\
&D(\pi_1 \ldots \pi_n) &&= D(\pi_1) \cup \ldots \cup D(\pi_n) \\
&D(<\pi_1\delta_1 \ldots \pi_a\delta_a \Diamond \pi>) &&= D(\pi_1) \cup \ldots \cup D(\pi_a) \cup D(\pi) \\
& &&\quad \cup D(\delta_1) \cup \ldots \cup D(\delta_a)
\end{aligned}
$$ □

Notice that by this definition of D, the copy action $y \gg x$ with y and x distinct is considered a definition as mentioned above. The preceding discussion justifies

Definition 2.4.1-2: The *interference* Interf(π) *of path* π is the set of variables interfering in π and is defined as follows

$$\mathrm{Interf}(\pi) = I(\pi)$$

where

$$
\begin{aligned}
&I:\ \mathrm{Path} \rightarrow \wp(\mathrm{Var}) \\
&I(\uparrow x) &&= \{\} \\
&I(\pi_1 \ldots \pi_n) &&= \{ x \mid \exists i,j.\ i<j \wedge x \in D(\pi_i) \wedge x \in U_0(\pi_j) \} \\
& &&\quad \cup I(\pi_1) \cup \ldots \cup I(\pi_n) \\
&I(<\pi_1\delta_1 \ldots \pi_a\delta_a \Diamond \pi>) &&= \{ x \mid \exists i,j.\ i<j \wedge x \in D(\pi_i) \cup D(\delta_i) \wedge x \in U_0(\pi_j) \} \\
& &&\quad \cup \{ x \mid \exists i,j.\ i<j \wedge (\delta_i = \downarrow x \text{ or } \delta_i = z \gg x \text{ for some } z) \\
& &&\qquad\qquad \wedge x \in D(\pi_j) \wedge x \in U_0(\pi) \} \\
& &&\quad \cup I(\pi_1) \cup \ldots \cup I(\pi_a) \cup I(\pi)
\end{aligned}
$$ □

Definition 2.4.1-3: The *interference* Interf(pgm) *of program* pgm is the set of variables interfering in any path of the program:

$$\mathrm{Interf}(\mathrm{pgm}) = \cup \{ \mathrm{Interf}(\pi) \mid \pi \in \Pi(\mathrm{pgm}) \}$$ □

Definition 2.4.1-4: Variable $x \in \mathrm{Var}$ is *interfering in program* pgm iff $x \in \mathrm{Interf}(\mathrm{pgm})$. □

We have seen that if x is not interfering in pgm then x is *globalizable* in pgm: it may be replaced by a global variable without modifying the meaning of the program. The correctness of this will be proved on the basis of the semantics and path semantics for L programs in Section 3.4, but the definition of interference above should also make this intuitively plausible. To illustrate the definitions we consider a number of examples.

Example 2.4.1-1: Interference with respect to x, first case. Consider the du-path

$\uparrow x < \downarrow x \lozenge \ldots > \uparrow x$

which could correspond to evaluation of `x+g(2)+x` where `g(x)=...x+g(2)+x...` . This path has interference with respect to x because x becomes redefined in the call to g before its last use. Clearly, x could not be replaced by a global variable X as in `X+g([X:=2])+X` without risk of modifying the result. The path matches the second case of Definition 2.4.1-2 with $\pi_1 = \uparrow x$, $\pi_2 = < \downarrow x \lozenge \ldots >$ and $\pi_3 = \uparrow x$, and x is easily seen to be interfering in the path according to the definition. □

Example 2.4.1-2: Interference with respect to x, second case. Consider the du-path

$< \downarrow x \uparrow x \downarrow y \lozenge \ldots >$

which could correspond to evaluation of `g(2,x*3)` where `g(x,y)=...g(2,x*3)...` . This path has interference with respect to x because x becomes redefined in the first argument expression of the call to g before it is used in the second argument expression. Clearly, x could not be replaced by a global variable X as in `g([X:=2],X)` without risk of modifying the result. The path matches the third case of Definition 2.4.1-2 with $\pi_1 = \varepsilon$, the empty path, $\delta_1 = \downarrow x$, $\pi_2 = \uparrow x$, $\delta_2 = \downarrow y$ and $\pi = \ldots$, and x is easily seen to be interfering according to the definition. □

Example 2.4.1-3: Interference with respect to x, third case. Consider the du-path

$< \downarrow x < \downarrow x \downarrow y \lozenge \uparrow x > \downarrow y \lozenge \uparrow x >$

which could correspond to evaluation of `g(2,g(3,5))` where `g(x,y)=x`. This path has interference with respect to x because x becomes defined (intentionally) in the first argument expression of the outer call to g, but is redefined (unintentionally) in the nested call before it is used during evaluation of the body corresponding to the outer call. So x clearly cannot be replaced by a global variable X as in `g([X:=2],g([X:=3],5))`; this would evaluate to 3 whereas the original expression would evaluate to 2. The path matches the third case of Definition 2.4.1-2 with $\pi_1 = \varepsilon$, $\delta_1 = \downarrow x$, $\pi_2 = < \downarrow x \downarrow y \lozenge \uparrow x >$, $\delta_2 = \downarrow y$ and $\pi = \uparrow x$, and x is easily seen to be interfering according to the definition. □

Example 2.4.1-4: No interference with respect to x. The du-path

$< < \downarrow x \lozenge \uparrow x > \downarrow y \lozenge \ldots >$

corresponding to evaluation of `g(f(x*2))` where `f(x)=x*7` and `g(y)=...` has no interference with respect to x, for there cannot be any level 0 uses of x in ..., the body of g. □

Notice that if the delimiters "<", "◊", and ">" were dropped, then this path would be identical to the one in Example 2.4.1-2 which has interference for x. From this we conclude that these delimiters really convey useful information for the detection of interference.

Example 2.4.1-5: No interference in the powers of two algorithm. Consider the du-path for evaluation of the powers of two algorithm shown in Example 2.3.1-1:

$<\downarrow r \downarrow n \lozenge \uparrow n < \uparrow r \downarrow r \uparrow n \downarrow n \lozenge \uparrow n \uparrow r >>$

None of r and n is interfering in this path (as is verified using Definition 2.4.1-2) which shows that this particular evaluation would not be disturbed if r and n were replaced by global variables. This is in accordance with our claim in Example 2.1-1 that r and n could be replaced by global variables without disturbing *any* evaluation of the program. □

Example 2.4.1-6: The μP interpreter. Consider the du-path shown in Example 2.3.1-2 for evaluation of the μP interpreter on source program "a:=1+3". This path has no interference with respect to c, sc, and se, but it has with respect to e. First e is defined in line 2, then redefined in line 3 before the uses in line 5 that correspond to the first definition. This case of interference is the same as that in Example 2.4.1-1 above.

The path given in Example 2.3.1-3 for evaluation of the interpreter on source program "skip;skip" has no interference for sc, e, and se, but it has for c. A value for c is defined and then changed in line 2 before its last use in line 5. This case of interference is the same as that in Example 2.4.1-3 above. We conclude from these two paths that neither c nor e could be replaced by a global variable without changing the value of some evaluation, and that sc and se may be globalized at least in these example evaluations. Later on we shall see that sc and se may in fact be replaced by one common global variable in all evaluations. □

We have defined the interference Interf(π) of a du-path π as the set of variables that cannot be replaced by global variables without possibly disturbing the evaluation corresponding to π. In the next section we shall see how to detect whether two or more global variables can be replaced by the same global variable.

2.4.2 Variable Groups

In some cases not only a collection of local variables can each be replaced by a global variable, they can even be replaced by a common global variable. Detection of such cases calls for a refinement of the interference concept from the preceding section.

In the path ...$\downarrow x \uparrow y$..., neither x nor y are interfering, but it would be wrong to allocate them as the same global variable: the definition of x would affect y too (because they would share) and could change the value of their common global variable. We shall say that x *interferes with* y in this case. More precisely, x interferes with y in path π if and only if changing all definitions ($\downarrow x$) of x into definitions ($\downarrow y$) of y would make y interfering in π. Here a copy action z»x counts as a definition of x if z is distinct from x. Observe that x interferes with x in π precisely if x is interfering in π according to the definition in the preceding section.

Thus replacement of x and y by the same global variable may introduce interference, but it may also have the opposite effect. Variable y is interfering in path $...x»y \uparrow y...$, for it becomes defined by the copy action, assuming that variables x and y are distinct. But if they are replaced by the same global variable, then the copy action cannot modify the value of y and therefore y is not interfering in the path. This is the very reason for having the copy action in the language of du-paths: it allows us to recognize situations where copying the value of one variable to another is harmless.

We shall make the "interferes with" relation relative to the collections of variables that we intend to allocate together. The concepts of variable group and variable grouping are useful here. A *variable group* is a non-empty collection $\gamma \subseteq \mathrm{Var}$ of variables. A *variable grouping* is a set Γ of disjoint variable groups. We interpret a variable group γ as a collection of variables considered for replacement by one global variable called x_γ. A variable cannot be replaced by two different global variables at the same time, and therefore the variable groups in a variable grouping must be disjoint. Notice that a variable grouping is not necessarily a partition of the set of all variables: there may be variables that are not considered for allocation by a global variable. Formally,

$$\mathrm{VGroup} = \{\, \gamma \in \wp(\mathrm{Var}) \mid \gamma \neq \{\} \,\}$$
$$\mathrm{VGrouping} = \{\, \Gamma \subseteq \mathrm{VGroup} \mid \forall \gamma_1, \gamma_2 \in \Gamma.\ \gamma_1 = \gamma_2 \text{ or } \gamma_1 \cap \gamma_2 = \{\} \,\}.$$

The first step in defining the "interferes with" relation formally is to modify the D function from Definition 2.4.1-1 into a function DG which finds the set of variables with definitions in π relative to a variable grouping Γ. A copy $y»x$ is considered a definition of x if and only if there is no $\gamma \in \Gamma$ such that $x, y \in \gamma$. Hence a variable group plays the same role as did each single variable in Definition 2.4.1-1 of D above.

Definition 2.4.2-1: The set of variables $DG(\pi)\Gamma$ with definition in path π relative to Γ is defined as follows, where $\pi_i \in \mathrm{Path}$ and $\delta_i \in \mathrm{Def}$:

$$\begin{aligned}
&DG: (\mathrm{Path} \cup \mathrm{Def}) \to \mathrm{VGrouping} \to \wp(\mathrm{Var}) \\
&DG(\uparrow x)\Gamma && = \{\} \\
&DG(\downarrow x)\Gamma && = \{\, x \,\} \\
&DG(y»x)\Gamma && = \{\, x \mid \text{there is no } \gamma \in \Gamma \text{ such that } x \in \gamma \wedge y \in \gamma \,\} \\
&DG(\pi_1 \ldots \pi_n)\Gamma && = DG(\pi_1)\Gamma \cup \ldots \cup DG(\pi_n)\Gamma \\
&DG(<\pi_1\delta_1 \ldots \pi_a\delta_a \Diamond \pi>)\Gamma && = DG(\pi_1)\Gamma \cup \ldots \cup DG(\pi_a)\Gamma \cup DG(\pi)\Gamma \\
& && \quad \cup DG(\delta_1)\Gamma \cup \ldots \cup DG(\delta_a)\Gamma
\end{aligned}$$
□

Definition 2.4.2-2: The *interference* $\mathrm{Interf}(\pi,\Gamma)$ *of path* π relative to variable grouping Γ is

$$\mathrm{Interf}(\pi,\Gamma) = \mathrm{I}(\pi)\Gamma$$

where

$$\begin{array}{lll}
\mathrm{I}\colon \mathrm{Path} \to \mathrm{VGrouping} \to \wp(\mathrm{Var}^2) & & \\
\mathrm{I}(\uparrow x)\Gamma & = & \{\} \\
\mathrm{I}(\pi_1 \ldots \pi_n)\Gamma & = & \{ (x,y) \mid \exists i,j.\ i<j \wedge x \in \mathrm{DG}(\pi_i)\Gamma \wedge y \in \mathrm{U}_0(\pi_j) \} \\
 & & \cup\, \mathrm{I}(\pi_1)\Gamma \cup \ldots \cup \mathrm{I}(\pi_n)\Gamma \\
\mathrm{I}(<\pi_1\delta_1 \ldots \pi_a\delta_a \Diamond \pi>)\Gamma & = & \{ (x,y) \mid \exists i,j.\ i<j \wedge x \in \mathrm{DG}(\pi_i)\Gamma \cup \mathrm{DG}(\delta_i)\Gamma \wedge y \in \mathrm{U}_0(\pi_j) \} \\
 & & \cup\, \{ (x,y) \mid \exists i,j.\ i<j \wedge (\delta_i = \downarrow y \text{ or } \delta_i = z \gg y \text{ for some } z) \\
 & & \qquad \wedge\, x \in \mathrm{DG}(\pi_j)\Gamma \cup \mathrm{DG}(\delta_j)\Gamma \wedge y \in \mathrm{U}_0(\pi) \} \\
 & & \cup\, \mathrm{I}(\pi_1)\Gamma \cup \ldots \cup \mathrm{I}(\pi_a)\Gamma \cup \mathrm{I}(\pi)\Gamma \qquad \square
\end{array}$$

Choosing Γ to be the finest possible variable grouping $\Gamma_{\mathrm{finest}} = \{ \{x\} \mid x \in \mathrm{Var} \}$, this definition extends Definition 2.4.1-2: for all $\pi \in \mathrm{Path}$ and $x \in \mathrm{Var}$ it holds that $x \in \mathrm{Interf}(\pi)$ if and only if $(x,x) \in \mathrm{Interf}(\pi, \Gamma_{\mathrm{finest}})$.

Example 2.4.2-1: Consider the μP interpreter from Example 2.1-2. There we argued that variable `sc` of function `cmd` and variable `se` of function `exp` could be replaced by one global variable `s` without altering the meaning of the program. Indeed, none of (`sc`,`sc`), (`sc`,`se`), (`se`,`sc`), and (`se`,`se`) is in $\mathrm{Interf}(\pi,\Gamma)$ where π is the du-path in Example 2.3.1-2 and Γ = { {`sc`, `se`} }. □

Definition 2.4.2-3: The *interference* $\mathrm{Interf}(\mathrm{pgm},\Gamma)$ *of program* pgm relative to Γ is

$$\mathrm{Interf}(\mathrm{pgm},\Gamma) = \cup \{ \mathrm{Interf}(\pi,\Gamma) \mid \pi \in \Pi(\mathrm{pgm}) \}. \qquad \square$$

This extends Definition 2.4.1-3: for all $x \in \mathrm{Var}$ it holds that $x \in \mathrm{Interf}(\mathrm{pgm})$ if and only if $(x,x) \in \mathrm{Interf}(\mathrm{pgm}, \Gamma_{\mathrm{finest}})$. The interference of a program is a subset of Var^2, that is, a relation on the set of variables.

Definition 2.4.2-4: Variable group $\gamma \in \Gamma$ is *interfering in path* π relative to Γ iff there are x and y in γ such that $(x,y) \in \mathrm{Interf}(\pi,\Gamma)$. □

Definition 2.4.2-5: Variable group $\gamma \in \Gamma$ is *interfering in program* pgm relative to Γ iff there is a path $\pi \in \Pi(\mathrm{pgm})$ such that γ is interfering in π relative to Γ. □

This extends the "interfering in" concept of Section 2.4.1: the singleton variable group { x } is interfering in pgm relative to Γ_{finest} precisely if variable x is interfering in pgm.

We extend the above definitions to apply to variable groupings also. Variable grouping Γ is interfering in path π iff there is a $\gamma \in \Gamma$ which is interfering in π relative to Γ. Variable grouping Γ is interfering in program pgm iff there is $\pi \in \Pi(\mathrm{pgm})$ such that Γ is interfering in π.

Intuitively, γ is interfering in pgm relative to Γ if there are x and y in γ and an evaluation (a path) in which a definition of x may change the current value of y before its last use, assuming that all variables in γ are allocated as the same global variable. Notice that x may be identical to y. Conversely, if γ is not interfering relative to Γ, then no definition of any variable x in γ can change the value of any y in γ before its last use, assuming that all variables in γ are allocated as one global variable. Hence it is safe to replace all the variables in γ by one and the same global variable.

This can be proved formally once the globalization transformation is made precise. The correctness proof involves the standard semantics and the path semantics for applicative programs, the standard semantics for imperative programs, and the above definitions of interference. Since all these components are precisely defined, the proof will be completely rigorous. The correctness of globalization for L programs is proved in Proposition 3.4-1, whereas the higher order case (H programs) is proved in Proposition 5.2-1.

In Section 2.4.1 we defined the interference Interf(π) of a path π without considering variable groups. In this section we have defined Interf(π,Γ) relative to a variable grouping, by letting each variable group $\gamma \in \Gamma$ play the same role (in copy actions) as single variables did before. The reader may find it strange that the first kind of interference is a set of variables, while the second is a relation on the set of all variables. Would it not be more natural to let Interf(π,Γ) be the set of those variable groups in Γ that are interfering in π relative to Γ? This approach is perfectly possible and would be useful for *detection* of interference. But we are interested also in *construction* of non-interfering variable groupings. Assume that we have tried some variable group γ and found that it is interfering in pgm relative to Γ. Then we must throw out some elements of γ. Using the information that x interferes with y, say, tells us that either x or y must out. If we did not have this information, we would have to try out all possible ways to throw out elements of γ. Section 3.3 presents an algorithm to construct a non-interfering variable grouping Γ_{non} for a given program.

Using the interference criteria given in Sections 2.4.1 and 2.4.2 we are able to make precise statements about interference in any given path (corresponding to a given evaluation). The path semantics presented in Section 2.3.2 enables us to make such statements not only for a given path, but also for a given L expression (or program) and a given initial environment: the path semantics supplies us with the path for the given program and initial environment, and we apply the criteria to that path.

Below we shall see how to make *safe approximate* statements about the interference of a program when only the program and no fixed initial environment is given. (A safe approximation to the interference relation Interf(pgm,Γ) is a relation that includes it). We do this by constructing a *definition-use grammar* that generates a superset of the set of paths for the program. Then we define an *interference analysis* that simulates the working of the I function from Definition 2.4.2-2 on grammar rules instead of paths. Applying the interference analysis to the grammar then produces an approximation to Interf(pgm,Γ).

3. DEFINITION-USE GRAMMARS: INTERFERENCE ANALYSIS

This section presents an analysis that for any given program determines a variable grouping which is definitely non-interfering. Also, a transformation is developed that will replace every variable group by a global variable. This section concerns the first order language L but most of the approach applies to the higher order case as well (with some further complications). Section 4 presents the extension to higher order languages.

In general, the set of possible paths for a given program (derivable by the path semantics) is infinite. Hence the path semantics cannot be used directly in an automatic analysis to detect possible interference. Section 3.1 shows how to construct a (finite context free) so-called *definition-use grammar* for a given program. This grammar approximates the path semantics in the sense that it generates a superset of the set of paths for the program. The finite grammar is then subjected to an automatic interference analysis that gives approximate but safe information about interference: if the approximate analysis says there is no interference, then it is guaranteed that there is none. On the other hand, the analysis may say that there is interference in cases where there is in fact none. Such an approximate *interference analysis* is developed in Section 3.2.

Section 3.3 presents an algorithm to find a non-interfering variable grouping for any given program. Section 3.4 presents the transformation that takes a non-interfering variable grouping and an L program, and produces an equivalent imperative L program where each variable group is replaced by one global variable. Finally, Section 3.5 gives an overview of concepts, analyses and transformations from Sections 2 and 3, placing each of the components in its proper context.

3.1 Definition-Use Grammars from Path Semantics

First, definition-use grammars are introduced and their use explained intuitively in Section 3.1.1. Then the grammar construction algorithm is developed in Section 3.1.2.

3.1.1 Definition-Use Grammars

A *definition-use grammar* (*du-grammar* for short) G_{pgm} for a given program pgm is a context free grammar that can generate every du-path of the program. In other words the du-grammar is a (finite and) safe approximation to the set of paths Π(pgm) of the program. The intention is that an analysis may be able to guarantee that a certain variable x is not interfering in any path generated by the *grammar*. Then we can conclude that x is not interfering in any path of the *program* with the implication that x is not interfering in pgm and may be globalized.

In this section we describe the principles of how to construct automatically a du-grammar for a given program from the path semantics. The principles apply not only to programs in the language L but to any language whose path semantics satisfies certain requirements.

The du-grammar G_{pgm} for program pgm has a distinguished start nonterminal N_{pgm} and other nonterminals N_e each corresponding to some (sub)expression e found in the program. The terminal symbols are those from which paths are constructed, that is, those of $\Sigma = \{ <, >, \Diamond \} \cup$ Use $\cup$ Def. (Notice that this set of symbols depends on the set Var of variables in pgm).

Whenever π is a path of pgm for some evaluation, π must be derivable from N_{pgm}, written $N_{pgm} \to^* \pi$. Similarly, if π is a path of expression e for some evaluation of e, then π shall be derivable from N_e, written $N_e \to^* \pi$. The idea behind the du-grammar construction is that a path semantics judgement

$$\rho \vdash e \Rightarrow v, \pi$$

may be taken to say that under certain circumstances, namely, in a certain environment ρ, evaluation of expression e may produce path π (in addition to some value v). Abstracting away from the circumstances (and the value v), we have the derivation

$$N_e \to^* \pi$$

expressing that evaluation of e may yield path π. Now consider an example expression `A(e1,e2)`. By disregarding ρ and the v's in path semantics rule PE2 for evaluation of `A(e1,e2)` and assuming arity(A) = 2, the rule

$$\frac{\begin{array}{l}\rho \vdash \mathtt{e1} \Rightarrow v_1, \pi_1 \\ \rho \vdash \mathtt{e2} \Rightarrow v_2, \pi_2\end{array}}{\rho \vdash \mathtt{A(e1,e2)} \Rightarrow v, \pi_1\pi_2} \qquad \begin{array}{l}\text{where } 2 = \text{arity}(\mathtt{A}) \\ \text{and } v = \mathtt{A}(v_1, v_2)\end{array}$$

abstracts to

$$\frac{\begin{array}{l}N_{\mathtt{e1}} \to^* \pi_1 \\ N_{\mathtt{e2}} \to^* \pi_2\end{array}}{N_{\mathtt{A(e1,e2)}} \to^* \pi_1\pi_2}.$$

In other words, if nonterminal N_{e1} can derive π_1 and N_{e2} can derive π_2, then $N_{\mathtt{A(e1,e2)}}$ can derive their concatenation $\pi_1\pi_2$. This is expressed more conventionally by the (context free) grammar rule

$$N_{\mathtt{A(e1,e2)}} \to N_{e1}N_{e2}.$$

This grammar rule then is an abstraction of (or approximation to) the path semantics rule instance just shown. Every grammar rule is an abstraction of a path semantics rule instance: a path semantics rule specialized to a concrete value for e. Grammar rules with nonterminals in the right hand side abstract path semantics rules with premises; those with only terminal symbols in the right hand side abstract path semantics rules without premises.

The du-grammar for a program must contain a sufficient set of grammar rules to approximate every path semantics rule instance used in some evaluation of the program. Importantly, in the case of the L path semantics this set is finite for any given program. Notice also that the idea behind this kind of grammar construction works only when the path semantics rules satisfy a rather natural requirement: the path in the conclusion must be a concatenation in some order of all the paths of the premises (and possibly further symbols). Otherwise the path semantics rules could not be approximated by context free grammar rules. All the path semantics rules of L satisfy this requirement as is easily verified.

Before proceeding to the actual construction of du-grammars we give a formal definition. For the purpose of this we denote the set of paths derivable from nonterminal N by $L(N) = \{ \pi \in \text{Path} \mid N \rightarrow^* \pi \}$. In particular $L(G_{pgm}) = L(N_{pgm})$ is the set of paths derivable by the grammar.

Definition 3.1.1-1: A *definition-use grammar* $G_{pgm} = (\Sigma, V_N, N_{pgm}, R)$ for program pgm with set of variables Var is a context free grammar with

- terminal symbols $\Sigma = \{ <, >, \Diamond \} \cup \text{Use} \cup \text{Def}$,
- nonterminal symbols $V_N \subseteq \{ N_{pgm} \} \cup \{ N_e \mid e \text{ is a (sub)expression in pgm} \}$,
- start nonterminal $N_{pgm} \in V_N$, and
- a set R of rules $N_e \rightarrow \alpha$, with rule right hand side $\alpha \in \text{Rhs}$ (defined below)

such that $\Pi(\text{pgm}) \subseteq L(N_{pgm})$. □

Alternatively, the last requirement may be stated like this: if $\pi \in \Pi(\text{pgm})$, then $N_{pgm} \rightarrow^* \pi$. The set Rhs of admissible grammar rule right hand sides α is defined inductively as follows:

$$\begin{aligned}\text{Rhs} = \; & V_N \\ & \cup \{ \uparrow x \mid x \in \text{Var} \} \\ & \cup \{ \alpha_1 \ldots \alpha_n \mid \alpha_j \in \text{Rhs}, j=1,\ldots,n;\; n \geq 0 \} \\ & \cup \{ <\alpha_1\delta_1 \ldots \alpha_a\delta_a \Diamond \alpha> \mid \alpha, \alpha_j \in \text{Rhs}, \delta_j \in \text{Def}, j=1,\ldots,a \}\end{aligned}$$

In particular we require that within each rule right hand side, function call structures are always balanced. Notice that $\text{Path} \subseteq \text{Rhs}$. We write $\alpha \rightarrow^* \pi$ when $\pi \in \text{Path}$ may be derived from $\alpha \in \text{Rhs}$, and $L(\alpha) = \{ \pi \in \text{Path} \mid \alpha \rightarrow^* \pi \}$ for the set of all paths derivable from α.

3.1.2 The Definition-Use Grammar Construction

Now we present the actual construction of a du-grammar for a given program. The grammar for pgm is constructed iteratively by adding new rules until every nonterminal used in the right hand side of a grammar rule is defined by one or more rules. While constructing the grammar we use the intention with each nonterminal N_e that if $\pi \in \Pi(e)$ then $N_e \rightarrow^* \pi$. This is *not* an invariant of the construction algorithm but a property of each nonterminal in the complete grammar that we shall prove later on (Proposition 3.1.2-1).

Algorithm 3.1.2-1: Definition-use grammar construction for L.
Input: An applicative L program pgm = `def` $\{ f^i(x^i_1, ..., x^i_{arity(f^i)}) = e^i \}_{i \in I}$ `in` e^0.
Output: The set R of rules of the definition-use grammar G_{pgm} for pgm.

```
R := { $N_{pgm} \rightarrow N_{e^0}$ };
while there is a nonterminal that is used but not defined in R do
    choose such a nonterminal $N_e$;
    case e of
        variable x        : R := R ∪ { $N_x \rightarrow \uparrow x$ }
        A(e1,...,ea)      : R := R ∪ { $N_{A(e1,...,ea)} \rightarrow N_{e1} N_{e2} ... N_{ea}$ }
        e1→e2,e3          : R := R ∪ { $N_{e1\rightarrow e2,e3} \rightarrow N_{e1} N_{e2}$,
                                       $N_{e1\rightarrow e2,e3} \rightarrow N_{e1} N_{e3}$ }
        $f^i$(e1,...,ea)  : R := R ∪ { $N_{f^i(e1,...,ea)} \rightarrow <N_{e1}\delta_1 ... N_{ea}\delta_a \Diamond N_{e^i}>$ }
                                       where $\delta_j = \Delta(x^i_j, ej)$ for j=1,...,a,
                                       and $e^i$ is the body of function $f^i$.
    endcase
endwhile                                                                    □
```

Initially, the grammar G_{pgm} consists of only the rule for the start nonterminal. This must be able to derive every path of the program, and according to the path semantics (rule PP1) every path of the initial expression e^0 is a path of the program pgm (and *vice versa*, in fact). Hence every string derivable by N_{e^0} is derivable by N_{pgm} which justifies intuitively the grammar rule $N_{pgm} \rightarrow N_{e^0}$. Because of the *vice versa*, there are no more rules for the start nonterminal: every path of the program pgm comes from the initial expression e^0.

The rest of the grammar is constructed by repeatedly selecting a nonterminal N_e which occurs in the right hand side of a rule but which is not (yet) defined by any rule. The grammar construction is complete and terminates when there is no such nonterminal left.

Now let N_e be such a nonterminal in the right hand side of a rule. We must add one or more rules to define N_e so that for every $\pi \in \Pi(e)$, it holds that π is derivable from N_e. This is done by simulating the evaluation of e (abstracting away from environments and values) as hinted at in Section 3.1.1 above. The expression e is matched against the expressions in the conclusions of the path semantics rules, and for every matching rule a grammar rule right hand side for N_e is constructed. This right hand side is a sequence of terminal and nonterminal symbols corresponding to the path expression in the conclusion. Terminal symbols (such as "<", "◊", etc.) in the path expression remain terminal symbols, but every path fragment corresponding to evaluation of an expression (e', say) in a premise is replaced by the corresponding nonterminal $N_{e'}$. Consulting the relevant path semantics rule in Section 2.3.2 (this is strongly recommended) for each case of e (from N_e) we find that

- if e is x (a variable occurrence), then path semantics rule PE1 matches and the grammar rule $N_x \rightarrow \uparrow x$ is added to G_{pgm}.
- if e is `A(e1,...,ea)` (a call of a basic function), then rule PE2 matches and the grammar rule $N_{A(e1,...,ea)} \rightarrow N_{e1}N_{e2}...N_{ea}$ is added to G_{pgm}.
- if e is `e1→e2,e3` (a conditional expression), then rules PE3 and PE4 match and so two grammar rules: $N_{e1\rightarrow e2,e3} \rightarrow N_{e1}N_{e2}$ and $N_{e1\rightarrow e2,e3} \rightarrow N_{e1}N_{e3}$ are added to G_{pgm}.
- if e is f^i`(e1,...,ea)` (a call of a defined function), then rule PE5 matches and the grammar rule $N_{f^i(e1,...,ea)} \rightarrow <N_{e1}\delta_1 ... N_{ea}\delta_a \Diamond N_{e^i}>$ is added to G_{pgm}, where e^i is the body of the called function f^i and $\delta_j = \Delta(x^i_j, ej)$ is a copy y»x or a definition $\downarrow x^i_j$ according as ej is a variable y or not, j=1,...,a. (Function Δ was defined at the end of Section 2.3.2).

The process of repeatedly selecting a nonterminal N_e which is used but not defined, and then adding rules to the grammar to define it, continues until no such nonterminal can be found anymore. Every nonterminal (except N_{pgm}) corresponds to a subexpression of pgm, so the number of possible nonterminals in the du-grammar for a given program is finite. In every iteration all the rules for one nonterminal are added to the grammar and therefore the procedure must terminate.

It is easy to see why the grammar does not in general derive the *exact* set of paths of pgm. Consider the conditional expression `true→e2,e3`. Obviously, the only paths for this are those of `e2`. But the du-grammar will contain both the rules $N_{true\rightarrow e2,e3} \rightarrow N_{true}N_{e2}$ and $N_{true\rightarrow e2,e3} \rightarrow N_{true}N_{e3}$ and will be able to derive also the paths for `e3`, although they are not actually possible for the program.

When using the grammar construction algorithm, we shall assume that it simply ignores those variables that are input parameters of the program pgm. The reason is that they are read-only variables (and not members of Var). They cannot be replaced by global variables, so they are not interesting for interference analysis.

Example 3.1.2-1: Definition-use grammar for the powers of two algorithm.
When applied to the program in Example 2.1-1, the du-grammar construction proceeds as follows. First the start symbol is defined in terms of the nonterminal for the initial expression e^0:

$$N_{pgm} \quad \rightarrow N_{f(1,x)}.$$

The only nonterminal used but not defined is $N_{f(1,x)}$ representing a call of a defined function.

We therefore add one rule

$$N_{\mathtt{f(1,x)}} \rightarrow \langle \downarrow\mathtt{r}\downarrow\mathtt{n} \lozenge N_{\mathtt{f}} \rangle$$

which uses the nonterminal $N_{\mathtt{f}}$ corresponding to `f`'s body `n=0`$\rightarrow$`r,f(2*r,n-1)`. This is a conditional expression and is defined in turn by two rules

$$N_{\mathtt{f}} \rightarrow N_{\mathtt{n=0}}N_{\mathtt{r}}$$
$$N_{\mathtt{f}} \rightarrow N_{\mathtt{n=0}}N_{\mathtt{f(2*r,n-1)}}.$$

Now "=" is a basic function so $N_{\mathtt{n=0}}$ is defined in terms of two further nonterminals. Notice that N_0 derives only the empty path ε. Nonterminal $N_{\mathtt{r}}$ requires only one rule:

$$N_{\mathtt{n=0}} \rightarrow N_{\mathtt{n}}N_0$$
$$N_{\mathtt{n}} \rightarrow \uparrow\mathtt{n}$$
$$N_0 \rightarrow \varepsilon$$
$$N_{\mathtt{r}} \rightarrow \uparrow\mathtt{r}.$$

The rule for the remaining nonterminal $N_{\mathtt{f(2*r,n-1)}}$ represents a call of a defined function:

$$N_{\mathtt{f(2*r,n-1)}} \rightarrow \langle N_{\mathtt{2*r}}\downarrow\mathtt{r}N_{\mathtt{n-1}}\downarrow\mathtt{n} \lozenge N_{\mathtt{f}} \rangle$$

and requires several further nonterminals

$$N_{\mathtt{2*r}} \rightarrow N_2N_{\mathtt{r}}$$
$$N_2 \rightarrow \varepsilon$$
$$N_{\mathtt{n-1}} \rightarrow N_{\mathtt{n}}N_1$$
$$N_1 \rightarrow \varepsilon.$$

Now all nonterminals used have been defined and the grammar is complete. It may be simplified considerably, however, by eliminating ε-rules and by replacing some nonterminals by the terminal strings they derive. This yields the final grammar for the powers of two program:

$$N_{\mathrm{pgm}} \rightarrow \langle \downarrow\mathtt{r}\downarrow\mathtt{n} \lozenge N_{\mathtt{f}} \rangle$$
$$N_{\mathtt{f}} \rightarrow \uparrow\mathtt{n}\uparrow\mathtt{r}$$
$$N_{\mathtt{f}} \rightarrow \uparrow\mathtt{n} \langle \uparrow\mathtt{r}\downarrow\mathtt{r}\uparrow\mathtt{n}\downarrow\mathtt{n} \lozenge N_{\mathtt{f}} \rangle$$

The du-path shown in Example 2.3.1-1 is derivable from N_{pgm} by this grammar as expected. □

Example 3.1.2-2: Du-grammar for the µP interpreter (Example 2.1-2):

$$N_{\mathrm{pgm}} \rightarrow \langle\downarrow\mathtt{c}\downarrow\mathtt{sc} \lozenge N_{\mathtt{cmd}}\rangle$$
$$N_{\mathtt{cmd}} \rightarrow \uparrow\mathtt{c}\uparrow\mathtt{sc}$$
$$N_{\mathtt{cmd}} \rightarrow \uparrow\mathtt{c}\uparrow\mathtt{c}\uparrow\mathtt{sc}\uparrow\mathtt{c}\langle\uparrow\mathtt{c}\downarrow\mathtt{e}\uparrow\mathtt{sc}\ \mathtt{sc}\text{»}\mathtt{se} \lozenge N_{\mathtt{exp}}\rangle$$
$$N_{\mathtt{cmd}} \rightarrow \uparrow\mathtt{c}\uparrow\mathtt{c}\uparrow\mathtt{c}\langle\uparrow\mathtt{c}\downarrow\mathtt{c}\ \langle\uparrow\mathtt{c}\downarrow\mathtt{c}\uparrow\mathtt{sc}\ \mathtt{sc}\text{»}\mathtt{sc} \lozenge N_{\mathtt{cmd}}\rangle\downarrow\mathtt{sc} \lozenge N_{\mathtt{cmd}}\rangle$$
$$N_{\mathtt{exp}} \rightarrow \uparrow\mathtt{e}\uparrow\mathtt{e}$$
$$N_{\mathtt{exp}} \rightarrow \uparrow\mathtt{e}\uparrow\mathtt{e}\uparrow\mathtt{e}\uparrow\mathtt{se}$$
$$N_{\mathtt{exp}} \rightarrow \uparrow\mathtt{e}\uparrow\mathtt{e}\uparrow\mathtt{e}\langle\uparrow\mathtt{e}\downarrow\mathtt{e}\uparrow\mathtt{se}\ \mathtt{se}\text{»}\mathtt{se}\lozenge N_{\mathtt{exp}}\rangle\ \langle\uparrow\mathtt{e}\downarrow\mathtt{e}\uparrow\mathtt{se}\ \mathtt{se}\text{»}\mathtt{se}\lozenge N_{\mathtt{exp}}\rangle$$

Notice that the paths shown in Example 2.3.1-2 and 2.3.1-3 are derivable from N_{pgm} by this grammar. □

A good implementation of Algorithm 3.1.2-1 should avoid the generation of ε-rules and should not generate nonterminals and extra rules for variable uses, but insert the relevant terminal symbols instead. In fact, nonterminals need be generated only for the initial expresion (N_{pgm}), for each conditional, and for each function body. Notice that the rule right hand sides generated by the algorithm have the form required for elements of Rhs. For the interference analysis developed in Section 3.2, the order and multiplicity of adjacent use action occurrences such as $\uparrow c\uparrow c\uparrow sc\uparrow c$ are irrelevant. For example, this may be simplified to $\uparrow sc\uparrow c$, which could be done by the algorithm also.

The correctness of the grammar construction is the subject of Proposition 3.1.2-1 below. First a little terminology. An expression occurrence is *reachable* in program pgm if

- it is the initial expression e^0, or
- it is a subexpression of some reachable expression, or
- it is the body of a function called from a reachable expression.

It is easy to see that a du-grammar has a nonterminal N_e exactly for those e that have a reachable occurrence. Furthermore, if some evaluation of pgm involves evaluation of a subexpression e, then it must be reachable. The converse does not hold, witness the expression $\texttt{true}\rightarrow e_2, e_3$ in which e_3 is reachable (if the entire expression is) but will never be evaluated. Unreachable expression occurrences are wholly superfluous and can be removed. Henceforth we assume that programs do not contain any unreachable expressions.

Proposition 3.1.2-1: The grammar G generated by Algorithm 3.1.2-1 satisfies

$$\Pi(\text{pgm}) \subseteq L(G).$$

Proof: We show that for every expression e and environment ρ, if $\rho \vdash e \Rightarrow v,\pi$ for some $v \in$ Value, then $N_e \rightarrow^* \pi$. Then the proposition follows by letting e be the initial expression e^0, because rule $N_{pgm} \rightarrow N_{e^0}$ is in G. The proof will be by induction on the structure of the evaluation tree for $\rho \vdash e \Rightarrow v,\pi$. The induction hypothesis is that for all ρ,e,v,π,

$$\rho \vdash e \Rightarrow v,\pi \quad \text{implies} \quad N_e \rightarrow^* \pi$$

whenever the evaluation tree for $\rho \vdash e \Rightarrow v,\pi$ is a subtree of the tree under consideration.

Case $e \equiv x$ matches path semantics rule PE1: We must have $\pi = \uparrow x$. But the rule $N_x \rightarrow \uparrow x$ is in the grammar, and therefore $N_x \rightarrow^* \uparrow x$. (Here we exploit that x must have a reachable occurrence in pgm, cf. the discussion above).

Case $e \equiv A(e_1,\ldots,e_a)$ matches rule PE2 with arity(A) = a: We must have $\pi = \pi_1\ldots\pi_a$ where $\rho \vdash e_j \Rightarrow v_j,\pi_j$ for j=1,...,a. By the induction hypothesis, $N_{ej} \rightarrow^* \pi_j$ for j=1,...,a, and therefore $N_{A(e1,\ldots,ea)} \rightarrow^* \pi_1\ldots\pi_a$, for G contains the rule $N_{A(e1,\ldots,ea)} \rightarrow N_{e1}\ldots N_{ea}$.

Case $e \equiv e_1 \rightarrow e_2, e_3$ matches rule PE3 or PE4: For rule PE3 (when e_1 evaluates to true) we must have $\pi = \pi_1\pi_2$ where $\rho \vdash e_j \Rightarrow v_j,\pi_j$ for j=1,2. By the induction hypothesis, $N_{ej} \rightarrow^* \pi_j$ for j=1,2, and therefore $N_{e1\rightarrow e2,e3} \rightarrow^* \pi_1\pi_2$, for G contains the rule $N_{e1\rightarrow e2,e3} \rightarrow N_{e1}N_{e2}$. The other case (PE4) is similar.

Case $e \equiv \mathtt{f}^i(\mathtt{e}_1,\ldots,\mathtt{e}_a)$ matches rule PE5 with arity$(\mathtt{f}^i) = a$: We must have $\pi = \langle \pi_1\delta_1 \ldots \pi_a\delta_a \lozenge \pi' \rangle$ where $\rho \vdash e_j \Rightarrow v_j, \pi_j$ and $\delta_j = \Delta(x^i_j, e_j)$ for $j=1,\ldots,a$ and $\rho' \vdash e^i \Rightarrow v, \pi'$. By the induction hypothesis, $N_{e_j} \rightarrow^* \pi_j$ for $j=1,\ldots,a$, and $N_{e^i} \rightarrow^* \pi'$.
Hence $N_{\mathtt{f}^i(\mathtt{e1},\ldots,\mathtt{ea})} \rightarrow^* \langle \pi_1\delta_1 \ldots \pi_a\delta_a \lozenge \pi' \rangle$, for G contains the rule
$N_{\mathtt{f}^i(\mathtt{e1},\ldots,\mathtt{ea})} \rightarrow \langle N_{e1}\delta_1 \ldots N_{ea}\delta_a \lozenge N_{e^i} \rangle$.

This concludes the induction proof and hence the proof of the proposition. □

3.2 Interference Analysis on Grammars

We shall develop a computable *interference analysis* § = ia$(G_{pgm})\Gamma$ working on a du-grammar G_{pgm}. This is to be a *safe approximation* to the interference relation Interf(pgm,Γ) on Var, shown in Definition 2.4.2-3. We choose to approximate that relation because it generalizes the interference criteria for single variables from Definition 2.4.1-3. Moreover it will be particularly useful for determining a non-interfering variable grouping in Section 3.3 below.

The interference analysis ia is to be *safe* in the sense that

$$\text{ia}(G_{pgm})\Gamma \supseteq \text{Interf(pgm}, \Gamma).$$

Assume that for variable group $\gamma \in \Gamma$ and all $x, y \in \gamma$ it holds that $(x,y) \notin \text{ia}(G_{pgm})\Gamma$. Then $(x,y) \notin$ Interf(pgm,Γ) which shows that γ is not interfering in pgm relative to Γ. Hence γ is safely globalizable: all the variables in γ can be replaced by one common global variable.

On the other hand the analysis ia is *approximate*. It may well be the case that $(x,y) \in \text{ia}(G_{pgm})\Gamma$ and yet $(x,y) \notin$ Interf(pgm,Γ). So we may erroneously decide from the analysis that γ is not globalizable when in fact it is. The analysis is approximate because it works on the set of paths generated by the du-grammar, and in general this set may include some paths not actually possible for the program. An exact analysis would be incomputable.

We shall construct the interference analysis simply by "lifting" the function $I(\pi)\Gamma$ given in Definition 2.4.2-2 to work on a sequence α of grammar symbols instead of a single path π. A sequence α of grammar symbols may be taken to represent the set L(α) of paths derivable from α, and therefore in a sense I is lifted to work on sets of paths.

The interference analysis is developed in several steps. The definition of $I(\pi)\Gamma$ used the functions $U_0(\pi)$ and $DG(\pi)\Gamma$ to find the set of level zero uses in π and the set of definitions in π relative to Γ. Thus we must define "lifted" versions of these functions too. This is done in Section 3.2.1. The interference analysis is then defined in Section 3.2.2, and its correctness is proved in Section 3.2.3.

3.2.1 The Auxiliary Functions ua_0 and da_Γ

Consider lifting the U_0 function from Definition 2.4.1-1 to a function UA_0 that works on a sequence α of grammar symbols. Since α may contain a nonterminal N, UA_0 must be given information about the uses in any path π derivable from N. This information can be provided by a function uenv: $\mathrm{UEnv} = V_N \rightarrow \wp(\mathrm{Var})$. It must satisfy that uenv(N) is the set of variables with level 0 uses in any path derivable from nonterminal N.

We want $UA_0[\![\alpha]\!]uenv$ to be the set of variables with level 0 uses in any path π derivable from α, assuming that this set is uenv(N) for every nonterminal N. What requirements must we put on uenv? First, it must be consistent, so that uenv(N) contains $UA_0[\![\alpha]\!]uenv$ for every rule $N \rightarrow \alpha$ in G_{pgm}. Second, each set uenv(N) should contain no unnecessary variables (or else the result is too imprecise). Thus we define uenv to be the pointwise inclusion-least solution to the equation

$$uenv = \lambda N.\cup\{\ UA_0[\![\alpha]\!]uenv \mid \text{rule } N\rightarrow\alpha \text{ is in } G_{pgm}\ \}.$$

The formal definition of UA_0 is given below. For brevity, we further define $ua_0(\alpha)$ to be $UA_0[\![\alpha]\!]uenv$. For use in the interference analysis, $ua_0(\alpha)$ must be a safe approximation to the set of variables with a level zero use in path π for any π derivable from $\alpha \in$ Rhs. This is shown to hold in Lemma 3.2.3-2. The structural similarity between the definitions of U_0 and UA_0 should make the correctness intuitively plausible, however. Notice in particular that for $\alpha = \pi \in$ Path, $U_0(\pi) = UA_0[\![\alpha]\!]uenv$.

Definition 3.2.1-1: Given a du-grammar G_{pgm} we define ua_0: Rhs $\rightarrow \wp(\mathrm{Var})$ by

$$ua_0(\alpha) = UA_0[\![\alpha]\!]uenv$$

where uenv: $\mathrm{UEnv} = V_N \rightarrow \wp(\mathrm{Var})$ is the (pointwise inclusion-least) solution to

$$uenv = \lambda N.\cup\{\ UA_0[\![\alpha]\!]uenv \mid \text{rule } N\rightarrow\alpha \text{ is in } G_{pgm}\ \}$$

and UA_0: Rhs $\rightarrow$ UEnv $\rightarrow \wp(\mathrm{Var})$

$$\begin{aligned}
UA_0[\![N]\!]ue &= ue(N)\\
UA_0[\![\uparrow x]\!]ue &= \{x\}\\
UA_0[\![\alpha_1...\alpha_n]\!]ue &= UA_0[\![\alpha_1]\!]ue \cup ... \cup UA_0[\![\alpha_n]\!]ue\\
UA_0[\![<\alpha_1\delta_1...\alpha_a\delta_a\Diamond\alpha>]\!]ue &= UA_0[\![\alpha_1]\!]ue \cup ... \cup UA_0[\![\alpha_a]\!]ue
\end{aligned}$$
□

Definition 3.2.1-2: Given du-grammar G_{pgm} and variable grouping Γ, we define da_Γ: (Rhs$\cup$Def) $\rightarrow \wp(\mathrm{Var})$ by

$$da_\Gamma(\alpha) = DA[\![\alpha]\!]\Gamma\ denv_\Gamma$$

where $denv_\Gamma$: $\mathrm{DEnv} = V_N \rightarrow \wp(\mathrm{Var})$ is the (pointwise inclusion-least) solution to

$$denv_\Gamma = \lambda N.\cup\{\ DA[\![\alpha]\!]\Gamma\ denv_\Gamma \mid \text{rule } N\rightarrow\alpha \text{ is in } G_{pgm}\ \}$$

and DA: (Rhs$\cup$Def) $\rightarrow$ VGrouping $\rightarrow$ DEnv $\rightarrow \wp(\mathrm{Var})$

$$\begin{aligned}
DA[\![N]\!]\Gamma\ de &= de(N)\\
DA[\![\uparrow x]\!]\Gamma\ de &= \{\}\\
DA[\![\downarrow x]\!]\Gamma\ de &= \{\ x\ \}\\
DA[\![y \gg x]\!]\Gamma\ de &= \{\ x \mid \text{there is no } \gamma \in \Gamma \text{ such that } y, x \in \gamma\ \}\\
DA[\![\alpha_1...\alpha_n]\!]\Gamma\ de &= DA[\![\alpha_1]\!]\Gamma\ de \cup ... \cup DA[\![\alpha_n]\!]\Gamma\ de\\
DA[\![<\alpha_1\delta_1...\alpha_a\delta_a\Diamond\alpha>]\!]\Gamma\ de &= DA[\![\alpha_1]\!]\Gamma\ de \cup ... \cup DA[\![\alpha_a]\!]\Gamma\ de \cup DA[\![\alpha]\!]\Gamma\ de\\
&\quad \cup DA[\![\delta_1]\!]\Gamma\ de \cup ... \cup DA[\![\delta_a]\!]\Gamma\ de
\end{aligned}$$
□

This "lifted" version DA of DG from Definition 2.4.2-1 is developed by a reasoning completely similar to that for UA_0. For use in the interference analysis, the function $da_\Gamma(\alpha) = DA[\![\alpha]\!]\Gamma\ denv_\Gamma$ must safely approximate DG. This is formalized in Lemma 3.2.3-2 also.

3.2.2 The Approximate Interference Analysis

The development of a "lifted" version IA of the I function from Definition 2.4.2-2 is completely similar to that of the UA_0 function above. We want that $IA[\![\alpha]\!]\Gamma\ ienv_\Gamma$ is the interference in all paths derivable from α, assuming that $ienv_\Gamma(N)$ is the interference in all paths derivable from nonterminal N. As for UA_0, $ienv_\Gamma$ is the pointwise inclusion-least consistent element of $IEnv = V_N \rightarrow \wp(Var^2)$.

By putting $ia(G_{pgm})\Gamma = IA[\![N_{pgm}]\!]\Gamma\ ienv_\Gamma$ we then have the wanted safe interference analysis. Every path π of the program is derivable from N_{pgm}, so if there is a path π of pgm with $(x,y) \in Interf(\pi,\Gamma)$, then $(x,y) \in IA[\![N_{pgm}]\!]\Gamma\ ienv_\Gamma$ also. This should motivate the following definition. The correctness is proved in Proposition 3.2.3-1 below.

Definition 3.2.2-1: Let a du-grammar G_{pgm} and a variable grouping Γ for program pgm be given. The approximate interference analysis ia is defined by

$$ia(G_{pgm})\Gamma = IA[\![N_{pgm}]\!]\Gamma\ ienv_\Gamma$$

where $ienv_\Gamma$: $IEnv = V_N \rightarrow \wp(Var^2)$ is the (pointwise inclusion-least) solution to

$$ienv_\Gamma = \lambda N. \cup\{\ IA[\![\alpha]\!]\Gamma\ ienv_\Gamma \mid \text{rule } N\rightarrow\alpha \text{ is in } G_{pgm}\},$$

and

$$
\begin{aligned}
&IA: Rhs \rightarrow VGrouping \rightarrow IEnv \rightarrow \wp(Var^2)\\
&IA[\![N]\!]\Gamma\ ienv &&= ienv(N)\\
&IA[\![\uparrow x]\!]\Gamma\ ienv &&= \{\}\\
&IA[\![\alpha_1...\alpha_n]\!]\Gamma\ ienv &&= \{\ (x,y) \mid \exists i,j.\ i<j \wedge x \in da_\Gamma(\alpha_i) \wedge y \in ua_0(\alpha_j)\}\\
& &&\quad \cup IA[\![\alpha_1]\!]\Gamma\ ienv \cup ... \cup IA[\![\alpha_n]\!]\Gamma\ ienv\\
&IA[\![<\alpha_1\delta_1...\alpha_a\delta_a\lozenge\alpha>]\!]\Gamma\ ienv &&= \{\ (x,y) \mid \exists i,j.\ i<j \wedge x\in da_\Gamma(\alpha_i)\cup da_\Gamma(\delta_i) \wedge y\in ua_0(\alpha_j)\}\\
& &&\quad \cup \{\ (x,y) \mid \exists i,j.\ i<j \wedge (\delta_i{=}\downarrow y \text{ or } \delta_i{=}z\text{»}y \text{ for some } z)\\
& &&\qquad\qquad \wedge x\in da_\Gamma(\alpha_j) \cup da_\Gamma(\delta_j) \wedge y\in ua_0(\alpha)\ \}\\
& &&\quad \cup IA[\![\alpha_1]\!]\Gamma\ ienv \cup ... \cup IA[\![\alpha_a]\!]\Gamma\ ienv\\
& &&\quad \cup IA[\![\alpha]\!]\Gamma\ ienv \qquad \square
\end{aligned}
$$

The relation $\sqsubseteq$ on IEnv defined by

$$ienv_1 \sqsubseteq ienv_2 \ \text{ iff } \ \forall N\in V_N.\ ienv_1(N) \subseteq ienv_2(N)$$

is a partial ordering called the pointwise inclusion ordering. With this ordering, IEnv is a lattice with finite height (Lemma 3.2.3-3). Therefore a particularly straightforward way to compute $ienv_\Gamma$ is to iterate the monotonic functional F_Γ (shown at the end of Proposition 3.2.3-1) on approximations to $ienv_\Gamma$ starting with the least element $\perp_{Ienv} = \lambda N.\{\}$:

Algorithm 3.2.2-1: Computation of the fixed point ienv_Γ of F_Γ.
Input: A du-grammar G_{pgm} and a variable grouping Γ.
Output: $\text{ienv}_\Gamma \in \text{IEnv}$ satisfying $\text{ienv}_\Gamma(N) \supseteq \cup\{ \text{Interf}(\pi,\Gamma) \mid N \to^* \pi \}$ for every $N \in V_N$.

```
ienv := $\lambda N.\{\}$;
$\text{ienv}_{\text{new}}$ := $F_\Gamma(\text{ienv})$;
while ienv $\neq \text{ienv}_{\text{new}}$ do
    ienv := $\text{ienv}_{\text{new}}$;
    $\text{ienv}_{\text{new}}$ := $F_\Gamma(\text{ienv})$
endwhile;
$\text{ienv}_\Gamma$ := ienv
```
□

This algorithm is potentially exceedingly slow, however. The height of the lattice $(\text{IEnv},\sqsubseteq)$ is n^{2m} where n is the number of variables in Var and m is the number of nonterminals in the grammar. Thus *a priori* we have no better bound on the number of iterations than this number which is monstrous for all but the most trivial programs. However, an algorithm which makes explicit use of the dependencies among the nonterminals should be feasible and would have a much better worst-case behaviour.

The approximate interference analysis depends only on the du-grammar, not on the way the grammar is constructed. Therefore the interference analysis will be usable not only for L programs but also for the higher order language H: the criteria for interference in a path are the same for H as for L.

The next subsection (3.2.3) presents the correctness proposition for the interference analysis (and its proof), and may be skipped except for the proposition. An example use of the interference analysis in finding a non-interfering variable grouping appears in Section 3.3 below.

3.2.3 Correctness of the Interference Analysis

Proposition 3.2.3-1: For any given du-grammar G_{pgm} and variable grouping Γ for pgm it holds that

$$\text{ia}(G_{\text{pgm}})\Gamma \supseteq \text{Interf}(\text{pgm}, \Gamma)$$

and $\text{ia}(G_{\text{pgm}})\Gamma$ is effectively computable.

Proof: By Lemma 3.2.3-1 it holds that $\text{ienv}_\Gamma(N_{\text{pgm}}) \supseteq \cup\{ \text{Interf}(\pi,\Gamma) \mid N_{\text{pgm}} \to^* \pi \}$, and by Proposition 3.1.2-1 it holds that $\{ \pi \in \text{Path} \mid N_{\text{pgm}} \to^* \pi \} \supseteq \Pi(\text{pgm})$ which shows the first postulate. The second one follows from the fact (Lemma 3.2.3-3) that $(\text{IEnv},\sqsubseteq)$ is a (complete) lattice of finite height with least element $\perp_{\text{Ienv}} = \lambda N.\{\}$ such that

$$F_\Gamma = \lambda\text{ienv}.\ \lambda N.\ \cup\ \{ \text{IA}[\![\alpha]\!]\Gamma\ \text{ienv} \mid \text{rule } N \to \alpha \text{ is in } G_{\text{pgm}} \}$$

is monotonic (Lemma 3.2.3-4). Then the fixed point ienv_Γ of F_Γ is reached in finitely many steps by iteration of F_Γ on $\perp_{\text{Ienv}}$; it holds that $\text{ienv}_\Gamma = F_\Gamma{}^n(\perp_{\text{Ienv}})$ for some nonnegative integer n. □

Lemma 3.2.3-1: For the fixed point ienv_Γ of F_Γ it holds for every $N \in V_N$ that

$$\text{ienv}_\Gamma(N) \supseteq \bigcup\{ \text{Interf}(\pi,\Gamma) \mid N \to^* \pi \}.$$

Proof: It is sufficient to prove that for each $\alpha \in \text{Rhs}$ and $\pi \in \text{Path}$ with $\alpha \to^* \pi$,

$$\text{IA}[\![\alpha]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{Interf}(\pi,\Gamma)$$

for then $\text{ienv}_\Gamma(N) = \bigcup\{ \text{IA}[\![\alpha]\!]\Gamma\ \text{ienv}_\Gamma \mid \text{rule } N\to\alpha \text{ is in } G_{pgm} \} \supseteq$
$\bigcup\{ \text{Interf}(\pi,\Gamma) \mid \alpha \to^* \pi \text{ where rule } N\to\alpha \text{ is in } G_{pgm}\} = \bigcup\{ \text{Interf}(\pi,\Gamma) \mid N \to^* \pi\}$.

The new thesis in turn follows by showing that

$$\text{IA}[\![\pi]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{Interf}(\pi,\Gamma) \qquad \text{for all } \pi \in \text{Path}$$

and

$$\text{IA}[\![\alpha]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{IA}[\![\alpha']\!]\Gamma\ \text{ienv}_\Gamma \qquad \text{whenever } \alpha \to^* \alpha' \text{ with } \alpha, \alpha' \in \text{Rhs},$$

and by taking $\alpha' = \pi \in \text{Path}$:

$$\text{IA}[\![\alpha]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{IA}[\![\pi]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{Interf}(\pi,\Gamma).$$

The first statement is easily proved by induction on the structure of π, using the correctness of auxiliary functions ua_0 and da_Γ (Lemma 3.2.3-2 below).

The second one is shown by induction on the length of the derivation $\alpha \to^* \alpha'$.

The base case is a derivation of length zero, that is, for $\alpha = \alpha'$, which is immediate.

For the inductive case, assume that $\text{IA}[\![\alpha]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{IA}[\![\alpha']\!]\Gamma\ \text{ienv}_\Gamma$ for all $\alpha, \alpha' \in \text{Rhs}$ for which $\alpha \to^* \alpha'$ in at most $m \geq 0$ steps (this is called the major induction hypothesis), and let $\alpha, \alpha'' \in \text{Rhs}$ such that $\alpha \to^* \alpha''$ in at most $m+1$ steps. We use induction on the structure of α. The base case is $\alpha = N$, a nonterminal; for this case there must be some $\alpha' \in \text{Rhs}$ such that rule $N\to\alpha'$ is in G_{pgm} and $\alpha' \to^* \alpha''$ in at most m steps. Then

$$\begin{aligned}\text{IA}[\![N]\!]\Gamma\ \text{ienv}_\Gamma &= \text{ienv}_\Gamma(N) = \\ &\bigcup\{ \text{IA}[\![\alpha''']\!]\Gamma\ \text{ienv}_\Gamma \mid \text{rule } N\to\alpha''' \text{ is in } G_{pgm} \} \\ &\supseteq \text{IA}[\![\alpha']\!]\Gamma\ \text{ienv}_\Gamma \\ &\supseteq \text{IA}[\![\alpha'']\!]\Gamma\ \text{ienv}_\Gamma,\end{aligned}$$

where the second "$\supseteq$" is because of the major induction hypothesis.

For the (first inductive) case $\alpha = \alpha_1...\alpha_n$, there are α''_i such that $\alpha'' = \alpha''_1... \alpha''_n$. There must be some j such that $\alpha_j \to^* \alpha'_j$ in one step. Then $\alpha'_j \to^* \alpha''_j$ in at most m steps, so $\text{IA}[\![\alpha'_j]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{IA}[\![\alpha''_j]\!]\Gamma\ \text{ienv}_\Gamma$ by the major induction hypothesis. Furthermore, by the minor induction hypothesis (for induction on α), $\text{IA}[\![\alpha_j]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{IA}[\![\alpha'_j]\!]\Gamma\ \text{ienv}_\Gamma$.
For $i \neq j$, $\alpha_i \to^* \alpha''_i$ in at most m steps, and so for all $i=1,...,n$ it holds that $\text{IA}[\![\alpha_i]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{IA}[\![\alpha''_i]\!]\Gamma\ \text{ienv}_\Gamma$. From this and the properties of ua_0 and da_Γ that $\text{ua}_0(\alpha') \supseteq \text{ua}_0(\alpha'')$ and $\text{da}_\Gamma(\alpha') \supseteq \text{da}_\Gamma(\alpha'')$ whenever $\alpha' \to^* \alpha''$, the desired conclusion follows: $\text{IA}[\![\alpha_1...\alpha_n]\!]\Gamma\ \text{ienv}_\Gamma \supseteq \text{IA}[\![\alpha''_1... \alpha''_n]\!]\Gamma\ \text{ienv}_\Gamma$.

For the other inductive case, $\alpha = \langle\alpha_1\delta_1...\alpha_a\delta_a \lozenge \alpha'\rangle$, the proof is similar. □

Lemma 3.2.3-2: For given du-grammar G_{pgm} and variable grouping Γ the functions ua_0 and da_Γ from Definitions 3.2.1-1 and 3.2.1-2 satisfy that for all $\alpha \in$ Rhs,

$$ua_0(\alpha) \supseteq \cup \{ U_0(\pi) \mid \alpha \rightarrow^* \pi \} \text{ and}$$
$$da_\Gamma(\alpha) \supseteq \cup \{ DG(\pi)\Gamma \mid \alpha \rightarrow^* \pi \}.$$

Proof: (Similar in structure to the proof of Lemma 3.2.3-1 above). For ua_0 it suffices to show that $ua_0(\pi) \supseteq U_0(\pi)$ for all $\pi \in$ Path, and that whenever $\alpha \rightarrow^* \alpha'$ with $\alpha, \alpha' \in$ Rhs, it holds that $ua_0(\alpha) \supseteq ua_0(\alpha')$. The first part is obvious from the definition of UA_0 because π contains no nonterminal symbols. The truth of the second part follows by induction on the length of the derivation $\alpha \rightarrow^* \alpha'$. The same proof procedure works for da_Γ. $\square$

Lemma 3.2.3-3: (IEnv,$\sqsubseteq$) is a lattice of finite height (and hence complete) with least element $\perp_{Ienv} = \lambda N.\{\}$ and greatest element $\top_{Ienv} = \lambda N.Var^2$.

Proof: Clearly, $\perp_{Ienv}$ is the least and $\top_{Ienv}$ is the greatest element of IEnv under the pointwise inclusion ordering $\sqsubseteq$. Then (IEnv,$\sqsubseteq$) is a lattice with least upper bounds $ienv_1 \sqcup ienv_2 = \lambda N.ienv_1(N) \cup ienv_2(N)$ and greatest lower bounds $ienv_1 \sqcap ienv_2 = \lambda N.ienv_1(N) \cap ienv_2(N)$ for all $ienv_1, ienv_2 \in$ IEnv. IEnv has finitely many elements, for the set Var of variables and the set V_N of nonterminals are finite, and then so are $\wp(Var^2)$ and $V_N \rightarrow \wp(Var^2)$. So trivially, (IEnv,$\sqsubseteq$) has finite height. $\square$

Lemma 3.2.3-4: The function F_Γ: IEnv $\rightarrow$ IEnv given by

$$F_\Gamma = \lambda ienv.\ \lambda N.\ \cup\{ IA[\![\alpha]\!]\Gamma\ ienv \mid \text{rule } N{\rightarrow}\alpha \text{ is in } G_{pgm}\}$$

is monotonic.

Proof: Let $ienv_1 \sqsubseteq ienv_2$ be given. It is sufficient to prove that for each $\alpha \in$ Rhs, $IA[\![\alpha]\!]\Gamma\ ienv_1 \subseteq IA[\![\alpha]\!]\Gamma\ ienv_2$. This is proved by induction on the structure of α:

α = nonterminal N: obvious from the definition of $\sqsubseteq$.

$\alpha = \uparrow x$: obvious that $IA[\![\alpha]\!]\Gamma ienv_1 = \{\} = IA[\![\alpha]\!]\Gamma ienv_2$.

$\alpha = \alpha_1...\alpha_n$: follows from the induction hypothesis for $\alpha_1, ..., \alpha_n$.

$\alpha = <\alpha_1\delta_1...\alpha_a\delta_a \lozenge \alpha'>$: follows from the induction hypothesis for $\alpha_1,..., \alpha_a$ and α' $\square$

With the correctness proposition and its lemmas the approximate interference analysis for du-grammars is complete. In the next section the interference analysis is employed in an algorithm to construct a non-interfering variable grouping for a given du-grammar (or program).

3.3 Construction of a Non-Interfering Variable Grouping

So far we have introduced paths, path semantics, interference and variable groups. We have shown how to construct a definition-use grammar that approximates the set of paths for a program. Then we developed the interference analysis for grammars that allows us to discover interference in the corresponding program. Now we want to use these achievements in an algorithm to construct a variable grouping which is not interfering in the given program.

Recall that a variable grouping Γ is a set of disjoint sets of variables called variable groups. The intention with each variable group $\gamma \in \Gamma$ is that all the variables in γ are replaced by one common global variable X_γ. Clearly, we want as many variables as possible to be allocated globally, thus saving the time and space overhead of stack allocation. Letting card(A) denote the number of elements of the (finite) set A, the total number card($\cup\Gamma$) of variables in the variable groups of Γ should therefore be as high as possible. At the same time the number of global variables used should be as small as possible thus increasing the sharing of global storage; hence the number card(Γ) of variable groups should be as small as possible. We therefore express the quality of a variable grouping Γ (rather crudely) in terms of its *index* $\#\Gamma$ defined as $\#\Gamma = 2*\text{card}(\cup\Gamma) - \text{card}(\Gamma)$, that is, two times the total number of variables globalized minus the number of global variables required.

It is desirable to find a non-interfering variable grouping with as high an index as possible. However, we have found no way of guaranteeing this short of trying out all variable groupings to see which are not interfering, and selecting one of those with greatest index. This is impractical (for there are very many different variable groupings), and therefore the algorithm given here is not in general optimal. To see that there is no "best" non-interfering variable grouping, consider a program with three variables x, y, and z. It may happen that x and y can be grouped together, and x and z can be grouped together, but x, y, and z cannot be grouped together. The two possible variable groupings {{x,y}, {z}} and {{x,z}, {y}} both have index 4, and thus none is not better than the other with this simple quality measure.

The algorithm we shall give approximates the wanted non-interfering Γ_{non} "from above", always working with a Γ that globalizes at least as many variables as the final consistent Γ_{non}. It starts with the coarsest possible variable grouping $\Gamma = \{\text{Var}\}$ that has the highest possible index $\#\{\text{Var}\} = 2n-1$ where n = card(Var) is the number of variables in the program under consideration. This initial Γ represents the optimistic hope to globalize all variables using only one global variable. In most cases this hope is inconsistent with the (approximation to) interference in the program as computed by $\text{ia}(G_{pgm})\Gamma$, and a new approximation must be computed until a Γ_{non} is obtained that is consistent with $\text{ia}(G_{pgm})\Gamma_{non}$.

A new approximation is computed using interference analysis on the du-grammar as follows. Using the current Γ, the relation $\S = \text{ia}(G_{pgm})\Gamma$ on Var is computed. Whenever $(x,x) \in \S$, variable x may be interfering in pgm relative to Γ despite the optimistic Γ. Hence every x with $(x,x) \in \S$ must be disregarded from inclusion in the final Γ. For the remaining variables, we know that a variable group $\gamma \in \Gamma$ is globalizable only if for all x, $y \in \gamma$ it holds that $(x,y) \notin §$ and $(y,x) \notin §$. It is instructive to rephrase this in terms of graph colourings.

Recall that a *colouring* of an irreflexive directed graph (V,E) with vertex set V and edge set $E \subseteq \{ (a,b) \in V^2 \mid a \neq b \}$ is a surjective function $p: V \rightarrow Q$ such that whenever $(a,b) \in E$ it holds that $p(a) \neq p(b)$. The elements of Q are called *colours* and a colouring must assign different colours to every two adjacent vertices. A *minimal* colouring is one that uses as few colours as possible (that is, as small a set Q as possible). A *colour component* of the colouring is a set of form $p^{-1}(q) = \{ a \in V \mid p(a) = q \}$ for $q \in Q$. The colour components are non-empty and disjoint, and cover the vertex set V. That is, the set of colour components is a partition of V.

Now let (V,E) be the irreflexive directed graph with vertex set $V = \{ x \in Var \mid (x,x) \notin § \}$ and edge set $E = V^2 \cap §$ and let $p: V \rightarrow Q$ be a colouring of (V,E). The graph has an edge from x to y precisely if x is interfering with y in some path derivable from G_{pgm} (relative to the current Γ). Hence if variables x and y have the same colour, then x does not interfere with y, and y does not interfere with x. So all variables with the same colour constitute a variable group which is not interfering in any path derivable from G_{pgm} relative to Γ. The converse is also true: if γ is a variable group not interfering in any path derivable from G_{pgm}, then all members of γ can be given the same colour.

Therefore a good new approximation Γ_{new} to a consistent variable grouping could be constructed by choosing a colouring of the graph and letting Γ_{new} be the set of colour components.

This procedure of making new approximations is iterated until Γ stabilizes. To ensure that this happens, we shall always choose the colouring such that no two vertices (that is, variables) that had different colours in the previous iteration will get the same colour by the new colouring. Then the algorithm terminates in at most 2n iterations (where n = card(Var)), since each iteration decreases the index of Γ by at least one and the smallest possible index is $0 = \#\{\}$. The index is decreased by at least one in each step since the new colouring of the graph is chosen such that each colour component is a subset of a colour component of the previous graph. From this it follows that the index of the new Γ is strictly smaller than the index of the previous one as we shall see below.

Algorithm 3.3-1: Construction of a non-interfering variable grouping Γ for a program.
Input: A du-grammar G_{pgm} and the set Var of variables of pgm.
Output: A variable grouping Γ which is not interfering in any π derivable from G_{pgm}.

```
$\Gamma$ := {Var};                                   - the coarsest variable grouping
$\Gamma_{new}$ := NewGamma($\Gamma$);
while  $\Gamma \neq \Gamma_{new}$ do
    $\Gamma$ := $\Gamma_{new}$;
    $\Gamma_{new}$ := NewGamma($\Gamma$)
endwhile;
```

where

```
function NewGamma($\Gamma$: VGrouping): VGrouping;
    § := ia($G_{pgm}$)$\Gamma$;                        - compute the relation § $\subseteq$ $Var^2$
    V := { $x \in$ Var | $(x,x) \notin$ § };              - construct the directed graph (V,E)
    E := $V^2 \cap$ §;
    let p: V $\rightarrow$ Q be a colouring of (V,E) such that p(x)=p(y) implies $x,y \in \gamma \in \Gamma$;
    $\Gamma_{new}$ := { $p^{-1}(q)$ | $q \in Q$ };
    return $\Gamma_{new}$
endfunction;                                                                    □
```

The correctness and termination properties of this algorithm are summarized in

Proposition 3.3-1: The above procedure terminates and the final Γ is not interfering in pgm.
Proof: The procedure must terminate, for in every iteration the index $\#\Gamma$ decreases by at least 1 (Lemma 3.3-2) and obviously $\#\Gamma$ must be nonnegative. Since Γ = NewGamma(Γ) by construction, it holds for every $\gamma \in \Gamma$ and $x, y \in \gamma$ that x and y have the same colour. Hence there is no edge (x,y) or (y,x) between x and y, and therefore $(x,y) \notin \text{ia}(G_{pgm})\Gamma$ and $(y,x) \notin \text{ia}(G_{pgm})\Gamma$. Then by Proposition 3.2.3-1, $(x,y) \notin \text{Interf}(\text{pgm},\Gamma)$ and $(y,x) \notin \text{Interf}(\text{pgm},\Gamma)$, which proves that γ is not interfering in pgm relative to Γ. Hence Γ is not interfering in pgm. □

Lemma 3.3-1: (1) The relation $\sqsubseteq$ on VGrouping defined by

$$\Gamma_1 \sqsubseteq \Gamma_2 \quad \text{iff} \quad \forall\gamma_1 \in \Gamma_1 \exists \gamma_2 \in \Gamma_2 .\ \gamma_1 \subseteq \gamma_2$$

is a partial ordering with least element { } and greatest element {Var}.

(2) The index function #: VGrouping $\rightarrow$ {0,1,2,...} defined by $\#\Gamma = 2*\text{card}(\cup\Gamma) - \text{card}(\Gamma)$ is strongly order preserving: if $\Gamma_1 \sqsubseteq \Gamma_2$ and $\Gamma_1 \neq \Gamma_2$ then $\#\Gamma_1 < \#\Gamma_2$.
Proof: (1): reflexivity, transitivity, and least and greatest elements are obvious, but for antisymmetry it is crucial that a variable grouping is a set of mutually disjoint subsets.
(2) The proof of strong order preservation is done by the two cases: $\cup\Gamma_1 = \cup\Gamma_2$ and $\cup\Gamma_1 \neq \cup\Gamma_2$. It depends heavily on the disjointness of the elements of Γ_1 and of Γ_2. The details are omitted. □

Lemma 3.3-2: In the algorithm, NewGamma(Γ) $\neq \Gamma$ implies #NewGamma(Γ) < #Γ.
Proof: This follows from (2) of Lemma 3.3-1 and from the fact that Γ_{new} is chosen such that $\Gamma_{new} \sqsubseteq \Gamma$ in NewGamma. □

Intuitively, the more variables that are grouped together by a variable grouping Γ, the fewer copy actions in α will be considered definitions by $da_\Gamma(\alpha)$, and the less interference should be found in α by $ia_\Gamma(a)$. That is, $\Gamma_1 \sqsubseteq \Gamma_2$ implies $da_{\Gamma 1}(\alpha) \supseteq da_{\Gamma 2}(\alpha)$. Using this property of da, we can further show that the more variables that are grouped together by Γ, the less variables interfere with each other: whenever $\Gamma_1 \sqsubseteq \Gamma_2$, it holds for every $\alpha \in$ Rhs that $ia_{\Gamma 1}(\alpha) \supseteq ia_{\Gamma 2}(\alpha)$. The result about da is proved below; that for ia can be proved by the same procedure. These results may be helpful when implementing the construction of a non-interfering Γ_{non}.

Lemma 3.3-3: If $\Gamma_1 \sqsubseteq \Gamma_2$ then for all $\alpha \in$ Rhs, DA$[\![\alpha]\!]\Gamma_1$ denv$_{\Gamma 1} \supseteq$ DA$[\![\alpha]\!]\Gamma_2$ denv$_{\Gamma 2}$ (where denv$_{\Gamma 1}$ and denv$_{\Gamma 2}$ are the fixed points from Definition 3.2.1-2).
Proof: First we make two observations

1. If $\Gamma_1 \sqsubseteq \Gamma_2$ then for all de $\in$ DEnv and $\alpha \in$ Rhs, DA$[\![\alpha]\!]\Gamma_1$ de $\supseteq$ DA$[\![\alpha]\!]\Gamma_2$ de. This is proved by induction on the structure of α where the interesting case is α = y»x.
2. If de$_2 \sqsubseteq$ de$_1$, then for every Γ and every $\alpha \in$ Rhs, DA$[\![\alpha]\!]\Gamma$ de$_1 \supseteq$ DA$[\![\alpha]\!]\Gamma$ de$_2$. This can be proved by induction on the structure of α; the interesting case is α = N.

Now, assuming $\Gamma_1 \sqsubseteq \Gamma_2$, we prove the lemma by proving DA$[\![\alpha]\!]\Gamma_1$ denv$_{\Gamma 1} \supseteq$ DA$[\![\alpha]\!]\Gamma_2$ de for the successive approximations de to the fixed point denv$_{\Gamma 2}$. Since denv$_{\Gamma 2}$ is reached in finitely many steps (for the lattice (DEnv,$\sqsubseteq$) has finite height), this proves the postulate. (This is a simple case of fixed point induction where the "admissibility" of the statement proved is obvious [Bird 1976]).

First we prove it for de = λN.{ }, the least element of DEnv. But for given $\alpha \in$ Rhs, DA$[\![\alpha]\!]\Gamma_1$ denv$_{\Gamma 1} \supseteq$ DA$[\![\alpha]\!]\Gamma_1$ λN.{ } $\supseteq$ DA$[\![\alpha]\!]\Gamma_2$ λN.{ } by observations 2 and 1.

Then under the assumption that for de $\in$ DEnv and for all $\alpha' \in$ Rhs, DA$[\![\alpha']\!]\Gamma_1$ denv$_{\Gamma 1}$ $\supseteq$ DA$[\![\alpha']\!]\Gamma_2$ de, we must prove that with de' = λN.$\cup${ DA$[\![\alpha'']\!]\Gamma_2$ de | rule N$\rightarrow\alpha''$ is in G$_{pgm}$ } the postulate holds for this new de' and for every $\alpha \in$ Rhs: DA$[\![\alpha]\!]\Gamma_1$ denv$_{\Gamma 1} \supseteq$ DA$[\![\alpha]\!]\Gamma_2$ de'.

So let $\alpha \in$ Rhs; the proof is done by induction on the structure of α, where the interesting case is α = N, a nonterminal. But

DA$[\![N]\!]\Gamma_1$ denv$_{\Gamma 1}$ =
denv$_{\Gamma 1}$(N) =
$\cup${ DA$[\![\alpha']\!]\Gamma_1$ denv$_{\Gamma 1}$ | N$\rightarrow\alpha'$ is in G$_{pgm}$ } $\supseteq$
$\cup${ DA$[\![\alpha']\!]\Gamma_2$ de | N$\rightarrow\alpha'$ is in G$_{pgm}$ } =
DA$[\![N]\!]\Gamma_2$ de

where the inclusion is because of the (major) induction hypothesis and the equalities hold because of the definition of DA. The other cases of α are simple. This completes the proof that DA$[\![\alpha]\!]\Gamma_1$ denv$_{\Gamma 1} \supseteq$ DA$[\![\alpha]\!]\Gamma_2$ denv$_{\Gamma 2}$ for all $\alpha \in$ Rhs. □

Example 3.3-1: Computation of Γ_{non} for the μP interpreter given in Example 2.1-2; the du-grammar G_{pgm} was shown in Example 3.1.2-2. The set of variables is Var = {c,sc,e,se}.

Initially, Γ_0 = {{c,sc,e,se}} which yields $§_0 = ia(G_{pgm})\Gamma_0$ = {(c,sc), (c,c), (e,e), (sc,c), (e,c), (e,se)}. The directed graph generated from this has vertex set V = {sc,se}, and empty edge set E = {}:

This clearly can be coloured using a single colour. The next approximation to Γ_{non} is Γ_1 = {{sc,se}}, and we get $§_1 = ia(G_{pgm})\Gamma_1$ which is identical to $§_0$ and so $\Gamma_{non} = \Gamma_1$ = {{sc,se}}. This shows that variables sc and se can be globalized using one global variable S as we informally concluded already in Example 2.1-2. □

In this section we have motivated and presented an algorithm that constructs a non-interfering variable grouping Γ_{non} for any program pgm when given its du-grammar G_{pgm}. The algorithm was proved to be correct. As to its efficiency, it requires at most 2n iterations to reach the wanted variable grouping, where n is the number of variables in pgm. Each iteration involves a recomputation of the interference analysis ia on the du-grammar, construction of a graph and finding a colouring of that graph. However, the construction of a *minimal* colouring is an NP-complete problem, and therefore no reasonably efficient algorithm for this is known [Aho, Hopcroft, Ullman 1974], [Garey, Johnson 1979]. Instead we may use so-called heuristic algorithms that yield approximate rather than minimal results. Such algorithms are discussed in for example [Brélaz 1979].

Also, the time for computing ia depends on the algorithm used; clearly a more efficient one than the proposed Algorithm 3.2.2-1 is desirable.

Notice that the mappings uenv, $denv_\Gamma$, and $ienv_\Gamma$ computed during the interference analysis need not be reconstructed from scratch in each iteration of Algorithm 3.3-1 above. This is because uenv is completely independent of the current approximation Γ to Γ_{non}, and in each iteration $\Gamma_{new} \sqsubseteq \Gamma$, so by Lemma 3.3-3, $denv_\Gamma \sqsubseteq denv_{\Gamma new}$ and $ienv_\Gamma \sqsubseteq ienv_{\Gamma new}$. Hence the old values $denv_\Gamma$ and $ienv_\Gamma$ are valid approximations to the new ones and can be used as starting points for finding these.

In the next section we shall use the non-interfering variable grouping Γ_{non} to control the globalization transformation of a given program.

3.4 The Globalization Transformation for L

This section presents the transformation T that replaces all non-interfering variables by global variables. The transformation takes as input an (applicative) L program pgm and a non-interfering variable grouping Γ and produces a transformed program pgm_Γ. This is an imperative L program which uses a global variable X_γ for each variable group $\gamma \in \Gamma$. To every function f^i in the original program pgm there corresponds a transformed function f^i in pgm_Γ.

The definition $f^i(x^i_1,..., x^i_a)=e^i$ of each function f^i is transformed by removing those parameters x^i_j that are to be globalized, that is, those for which there is a variable group $\gamma \in \Gamma$ such that $x^i_j \in \gamma$.

An expression e is transformed this way: every use of a variable x that belongs to some variable group $\gamma \in \Gamma$ is replaced by the global variable X_γ for that group. Every argument expression e_j in some position j of a call $f^i(e_1,..,e_j,..,e_a)$ of f^i is transformed as follows: if the variable x^i_j receiving the value of e_j belongs to some variable group $\gamma \in \Gamma$, then the argument expression e_j is replaced by a non-pushing assignment $[X_\gamma:=e_j]$ to the global variable X_γ for that group (and recursively, e_j is transformed also). The effect is that whenever pgm would evaluate argument expression e_j and push its value onto the parameter passing stack as a new value for variable x^i_j, the transformed program pgm_Γ will evaluate the (transformed) expression e_j and assign its value to the global variable X_γ, and hence push nothing onto the parameter passing stack.

Algorithm 3.4-1: The globalization transformation for L.
Input: An applicative L program

$$pgm = \texttt{def } \{f^i(x^i_1,..., x^i_{arity(f^i)})=e^i\}_{i\in I} \texttt{ in } e^0$$

and a variable grouping Γ for pgm which is not interfering in pgm.
Output: An imperative L program pgm_Γ in which every variable group $\gamma \in \Gamma$ is replaced by a global variable X_γ.
The transformed program is

$$pgm_\Gamma = \texttt{def } \{f^i(x^i_{j1},..., x^i_{jni})=te^i\}_{i\in I} \texttt{ in } te^0$$

where the variables $x^i_{j1},..., x^i_{jni}$ of transformed function f^i form a subsequence of the variables $x^i_1,..., x^i_a$ of the corresponding original function f^i. Variable x^i_j of original function f^i is among those of the transformed function if and only if there is no variable group $\gamma \in \Gamma$ such that x^i_j is in γ. The body expression e^i of function f^i is transformed into $te^i = T[\![e^i]\!]\Gamma$ where the expression transformation T is defined below. Similarly, the initial expression e^0 is transformed into $te^0 = T[\![e^0]\!]\Gamma$. □

Definition 3.4-1: The globalization transformation T for applicative L expressions is defined as follows (where LExpr is the set of applicative L expressions and LIExpr is the set of imperative ones):

$$\begin{array}{lll}
\multicolumn{3}{l}{T: \text{LExpr} \rightarrow \text{VGrouping} \rightarrow \text{LIExpr}} \\
T[\![\mathtt{x}]\!]\Gamma & = \mathtt{x}_\gamma & \text{if } \mathtt{x} \in \gamma \text{ for some } \gamma \in \Gamma \\
 & = \mathtt{x} & \text{otherwise} \\
T[\![\mathtt{A}(\mathtt{e}_1,\ldots,\mathtt{e}_a)]\!]\Gamma & = \mathtt{A}(T[\![\mathtt{e}_1]\!]\Gamma,\ldots,T[\![\mathtt{e}_a]\!]\Gamma) & \\
T[\![\mathtt{e}_1 \rightarrow \mathtt{e}_2, \mathtt{e}_3]\!]\Gamma & = T[\![\mathtt{e}_1]\!]\Gamma \rightarrow T[\![\mathtt{e}_2]\!]\Gamma, T[\![\mathtt{e}_3]\!]\Gamma & \\
T[\![\mathtt{f}^i(\mathtt{e}_1,\ldots,\mathtt{e}_a)]\!]\Gamma & = \mathtt{f}^i(\mathtt{e}''_1,\ldots,\mathtt{e}''_a) \text{ where} & \\
 & \quad \mathtt{e}''_j = [\mathtt{x}_\gamma := T[\![\mathtt{e}_j]\!]\Gamma] & \text{if } \mathtt{x}^i_j \in \gamma \text{ for } \gamma \in \Gamma \\
 & \quad \phantom{\mathtt{e}''_j} = T[\![\mathtt{e}_j]\!]\Gamma & \text{otherwise}
\end{array}$$

□

Note that in the last case (function call), if $\mathtt{e}_j \equiv \mathtt{x}_k \in \gamma$ and $\mathtt{x}^i_j \in \gamma$, then a trivial assignment $[\mathtt{x}_\gamma := \mathtt{x}_\gamma]$ would be generated. An improved transformation algorithm should avoid the generation of such trivial assignments.

The algorithm above takes a well-formed applicative L program pgm and a non-interfering variable grouping Γ and produces a well-formed imperative L program. The transformation implemented by the algorithm together with T is correct in the sense that pgm_Γ is *at least as strong as* pgm. More precisely, for every initial environment ρ_0, if pgm terminates with a result v, then pgm_Γ will also terminate with result v. This will be proved in Proposition 3.4-1 below. To prove that the two programs are *strongly equivalent*, we should prove also that if pgm does not terminate in initial environment ρ_0, then the transformed program pgm_Γ does not terminate either. For the present purposes, we consider this to be less important, though.

Example 3.4-1: The result of applying Algorithm 3.4-1 to the μP interpreter from Example 2.1-2 and the variable grouping $\Gamma = \{\{\mathtt{sc},\mathtt{se}\}\}$ found in Example 3.3-1 is the imperative program also shown in Example 2.1-2. The only difference is that $\mathtt{x}_{\{\mathtt{sc},\mathtt{se}\}}$ was called `S`, and trivial assignments `[S:=S]` were omitted. □

First we shall see how the correctness proof for the globalization transformation can be structured. We notice that the expression transformation function T does three things. First, it adds an assignment to global variable $\mathtt{x}_\gamma$ everywhere a new value of some variable x belonging to variable group $\gamma \in \Gamma$ is pushed onto the parameter passing stack. (This can happen only in an argument expression of a call to a defined function). Secondly, it replaces all uses of every variable x that belongs to a variable group γ by uses of the corresponding global variable $\mathtt{x}_\gamma$. Thirdly, it puts brackets around assignments $[\mathtt{x}_\gamma := \ldots]$ to global variables such that no value is pushed onto the parameter passing stack.

The transformation can be rewritten so that it works in precisely these three steps:

1: Add assignments: each function body is transformed so that whenever a new value for $x \in \gamma \in \Gamma$ is pushed onto the stack, the value pushed is also assigned to global variable X_γ.

2: Add uses of global variables: replace every use of $x \in \gamma \in \Gamma$ by use of X_γ.

3: Remove pushes: add brackets around assignments to global variables and remove from the definition of function f^i those parameters $x^i{}_j$ that belong to some $\gamma \in \Gamma$.

To these steps there correspond three expression transformation functions T_1, T_2, and T_3 (shown in Definition 3.4-2 below). Now it is easy to see why the total transformation is correct. Let an applicative L program pgm_0 be given and let pgm_1 be the transformed program after step 1 and so on. Then pgm_1 obviously produces the same result as pgm_0, for pgm_1 does the same actions as pgm_0 and in addition some assignments to global variables. It never references any global variables and so its evaluation does not depend on the value of these global variables.

But at any time a variable x that belongs to a variable group $\gamma \in \Gamma$ is needed during evaluation of pgm_1, the current value of the corresponding global variable X_γ equals that of x. This property will be called *consistency* of the evaluation of pgm_1. It will be formalized below as a property of the evaluation trees for pgm_1, and it will be proved in Proposition 3.4-1 that every evaluation tree of pgm_1 is consistent (if Γ is not interfering in pgm_0).

The importance of consistency is that every use of a variable x that belongs to variable group $\gamma \in \Gamma$ can be replaced by a use of the global variable X_γ: at every use of x, the global variable X_γ has the same value as x. This replacement is done by step 2, yielding program pgm_2 which therefore produces the same result as pgm_1.

But now in pgm_2 there are no more uses of any variable x that belongs to some $\gamma \in \Gamma$; the uses have been replaced by uses of the global variables. Thus we can safely delete all pushes of a new value for such variables x. This is done in step 3 by enclosing the argument expressions computing a new value for x in brackets [...]. (These argument expressions are precisely those that were made into assignments to X_γ in step 1). At the same time, each such x must be removed from the parameter list of the function definition in which it occurs; this is also done in step 3.

Clearly, the program pgm_3 obtained by this procedure produces the same result as pgm_2, and from this it follows that pgm_3 and the original pgm_0 are equivalent. Furthermore, it is easy to see that pgm_3 is identical to the pgm_Γ defined above, for T is the composition of T_1, T_2, and T_3 defined below. This proves the correctness of the transformation.

The rest of this section is a detailed proof that every evaluation of pgm_1 is consistent. The reader may skip to Proposition 3.4-2 at the end of the section without loss of continuity.

Definition 3.4-2: The expression transformation functions T_1, T_2, and T_3 are defined as follows.

$$
\begin{array}{lll}
T_1\text{: LExpr} \rightarrow \text{VGrouping} \rightarrow \text{LIExpr} & & \\
T_1[\![x]\!]\Gamma & = & x \\
T_1[\![A(e_1,\ldots,e_a)]\!]\Gamma & = & A(T_1[\![e_1]\!]\Gamma,\ldots, T_1[\![e_a]\!]\Gamma) \\
T_1[\![e_1\rightarrow e_2,e_3]\!]\Gamma & = & T_1[\![e_1]\!]\Gamma\rightarrow T_1[\![e_2]\!]\Gamma, T_1[\![e_3]\!]\Gamma \\
T_1[\![f^i(e_1,\ldots,e_a)]\!]\Gamma & = & f^i(e''_1,\ldots, e''_a) \text{ where} \\
 & & \quad e''_j = X_\gamma{:=}T_1[\![e_j]\!]\Gamma \quad \text{if } x^i_j \in \gamma \text{ for } \gamma \in \Gamma \\
 & & \quad\quad = T_1[\![e_j]\!]\Gamma \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
T_2\text{: LIExpr} \rightarrow \text{VGrouping} \rightarrow \text{LIExpr} & & \\
T_2[\![x]\!]\Gamma & = & X_\gamma \quad \text{if } x \in \gamma \text{ for some } \gamma \in \Gamma \\
 & = & x \quad \text{otherwise} \\
T_2[\![A(e_1,\ldots,e_a)]\!]\Gamma & = & A(T_2[\![e_1]\!]\Gamma,\ldots, T_2[\![e_a]\!]\Gamma) \\
T_2[\![e_1\rightarrow e_2,e_3]\!]\Gamma & = & T_2[\![e_1]\!]\Gamma\rightarrow T_2[\![e_2]\!]\Gamma, T_2[\![e_3]\!]\Gamma \\
T_2[\![f^i(e_1,\ldots,e_a)]\!]\Gamma & = & f^i(T_2[\![e_1]\!]\Gamma,\ldots, T_2[\![e_a]\!]\Gamma) \\
T_2[\![X_\gamma{:=}e]\!]\Gamma & = & X_\gamma{:=}T_2[\![e]\!]\Gamma
\end{array}
$$

$$
\begin{array}{lll}
T_3\text{: LIExpr} \rightarrow \text{LIExpr} & & \\
T_3[\![x]\!] & = & x \\
T_3[\![X_\gamma]\!] & = & X_\gamma \\
T_3[\![A(e_1,\ldots,e_a)]\!] & = & A(T_3[\![e_1]\!],\ldots, T_3[\![e_a]\!]) \\
T_3[\![e_1\rightarrow e_2,e_3]\!] & = & T_3[\![e_1]\!]\rightarrow T_3[\![e_2]\!], T_3[\![e_3]\!] \\
T_3[\![f^i(e_1,\ldots,e_a)]\!] & = & f^i(T_3[\![e_1]\!],\ldots, T_3[\![e_a]\!]) \\
T_3[\![X_\gamma{:=}e]\!] & = & [X_\gamma{:=}T_3[\![e]\!]]
\end{array}
$$
□

We still have to prove the consistency of every evaluation of pgm_1; this in turn requires a precise definition of consistency. We will apply this predicate only to terminating evaluations: those for which the evaluation tree is finite. An evaluation of program pgm is consistent if every judgement in the evaluation tree is consistent, where consistency of a judgement is defined below.

Definition 3.4-3: Let an (imperative) L program pgm be given. Expression evaluation judgement $\rho, \sigma \vdash e \Rightarrow v, \sigma'$ (in an evaluation tree for pgm) is *consistent* for variable grouping Γ iff whenever

- x belongs to variable group γ for some $\gamma \in \Gamma$,
- the current value (incarnation) of x is used in evaluation of e

it holds that

- $\rho(x) = \sigma(X_\gamma)$. □

Intuitively, a judgement is consistent if the value of x_γ equals that of $x \in \gamma \in \Gamma$ whenever the current incarnation of x is used in the evaluation of e. By the current incarnation of x we mean x as defined by the current local environment ρ. Notice that if the current incarnation of x is used in an evaluation of e, then there is a level zero use ($\uparrow$x) of x in the corresponding du-path for e (and vice versa). This can be verified by looking at the path semantics for L.

Proposition 3.4-1: (Correctness of the globalization transformation for L). Let an applicative L program pgm_0 be given and let Γ be a variable grouping which it not interfering in pgm_0. Then for every initial environment ρ_0,

if $\quad \rho_0 \vdash pgm_0 \Rightarrow v$

then $\quad \rho_0 \vdash pgm_\Gamma \Rightarrow v$

where pgm_Γ is the result of applying Algorithm 3.4-1 to pgm_0 and Γ, that is, pgm_0 with every variable group $\gamma \in \Gamma$ replaced by a global variable.

Proof: In accordance with the discussion of the correctness proof above, it is sufficient to show that every judgement in the evaluation tree for $\rho_0 \vdash pgm_1 \Rightarrow v$ is consistent, where pgm_1 is the result of applying expression transformation T_1 to every function body e^i and the initial expression e^0 in program pgm_0.

The proof is done by induction on the structure of the evaluation tree (which is finite). The lowest part of the evaluation tree looks like this (according to semantics rule IP1 from Section 2.2.3):

$$\cfrac{\cfrac{\cdots}{\rho_0, \sigma_{init} \vdash e^0 \Rightarrow v, \sigma'}}{\rho_0 \vdash pgm_1 \Rightarrow v}$$

where σ_{init} is the empty initial state of global variables and e^0 is the initial expression of pgm_1.

The base case is clear: since none of the input parameters in dom(ρ_0) is in Var, they are also not in any variable group $\gamma \in \Gamma$, and so the judgement $\rho_0, \sigma_{init} \vdash e^0 \Rightarrow v, \sigma'$ is trivially consistent.

The induction step is done by showing for every node (that is, branching point) of the tree that consistency of the judgement in the conclusion (below the line) implies consistency of all judgements in the premises (above the line). This is done by case of the form of expression e' in the conclusion judgement.

So let $\rho, \sigma_0 \vdash e' \Rightarrow v, \sigma_1$ be a consistent judgement in the tree. We can assume that $e' = T_1[\![e]\!]\Gamma$ where e is some (sub)expression from pgm_0, and further assume that $\rho \vdash e \Rightarrow v, \pi$ for some path π by the L path semantics. Now we consider the possible cases for expression e to prove that the premises of $\rho, \sigma_0 \vdash e' \Rightarrow v, \sigma_1$ are consistent.

Case $e \equiv x$: Then $e' \equiv x$ and there are no premises by semantics rule IE1; this is a leaf node in the tree.

Case $e \equiv \mathtt{A}(e_1,...,e_a)$: Then $e' \equiv \mathtt{A}(e'_1,...,e'_a)$. The path π is $\pi_1...\pi_a$ where $\rho \vdash e_j \Rightarrow v_j, \pi_j$ for j=1,...,a. The tree node under consideration is, by semantics rule IE2,

$$\frac{\rho, \sigma_{j-1} \vdash e'_j \Rightarrow v_j, \sigma_j \text{ for } j=1,...,a}{\rho, \sigma_0 \vdash \mathtt{A}(e'_1,...,e'_a) \Rightarrow v, \sigma_a}$$

and we prove the consistency of each $\rho, \sigma_{j-1} \vdash e'_j \Rightarrow v_j, \sigma_j$ by contradiction. Assume that $x \in \gamma \in \Gamma$, that x is used during evaluation of e'_j, and that $\rho(x) \neq \sigma_{j-1}(X_\gamma)$. Then evaluation of some e'_i with i<j must have modified X_γ, for its value was correct before evaluation of e' (by consistency of the conclusion judgement). But this can happen only by an assignment to X_γ during evaluation of e'_i, and therefore evaluation of the original e_i must involve a push of a new value for some variable y which belongs to the same variable group γ as x. But then $y \in DG(\pi_i)\Gamma$ and $x \in U_0(\pi_j)$ and so y interferes with x in π relative to Γ, which contradicts the assumption that Γ is not interfering in pgm_0. Hence $\rho, \sigma_{j-1} \vdash e'_j \Rightarrow v_j, \sigma_j$ must be consistent.

Case $e \equiv e_1 \rightarrow e_2, e_3$: Then $e' \equiv e'_1 \rightarrow e'_2, e'_3$. Assume that the conditional e_1 evaluates to true (the other case, false, is completely similar). Then the path $\pi = \pi_1\pi_2$ with $\rho \vdash e_j \Rightarrow v_j, \pi_j$ for j=1,2. The tree node under consideration is, by semantics rule IE3,

$$\frac{\rho, \sigma_0 \vdash e'_1 \Rightarrow \text{true}, \sigma_1 \qquad \rho, \sigma_1 \vdash e'_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash e'_1 \rightarrow e'_2, e'_3 \Rightarrow v, \sigma_2}$$

The consistency of both premises is proved in exactly the same manner as the above case.

Case $e \equiv \mathtt{f}^k(e_1,...,e_a)$, $k \in I$: Then $e' \equiv \mathtt{f}^k(e'_1,...,e'_a)$ where e'_j is $X_\gamma := e''_j$ if $x^k_j \in \gamma$ for some $\gamma \in \Gamma$. The path π is $<\pi_1\delta_1...\pi_a\delta_a \lozenge \pi'>$ where $\rho \vdash e_j \Rightarrow v_j, \pi_j$ for j=1,...,a; and $\rho' \vdash e^k \Rightarrow v, \pi'$ where $\rho'(x^k_j) = v_j$ for j=1,...,a.

The tree node under consideration is, by semantics rule IE5,

$$\frac{\begin{array}{l}\rho, \sigma_{j-1} \vdash e'_j \Rightarrow v_j, \sigma_j \text{ for } j=1,...,a \\ \rho, \sigma_a \vdash e'^k \Rightarrow v, \sigma\end{array}}{\rho, \sigma_0 \vdash \mathtt{f}^k(e'_1,...,e'_a) \Rightarrow v, \sigma} \qquad \text{where } \rho'(x^k_j) = v_j \text{ for } j=1,...,a$$

and $e'^k = T_1[\![e^k]\!]\Gamma$ is the transformed body of $\mathtt{f}^k$ in program pgm_1.

The consistency of each judgement $\rho, \sigma_{j-1} \vdash e'_j \Rightarrow v_j, \sigma_j$ is proved by contradiction. Assume variable $x \in \gamma \in \Gamma$ is used during evaluation of e'_j and $\rho(x) \neq \sigma_{j-1}(X_\gamma)$. Then evaluation of some e'_i with i<j must have assigned a new value to X_γ, for by consistency of the conclusion judgement, $\rho(x) = \sigma_0(X_\gamma)$. This can happen in two ways. Either e'_i is an assignment $X_\gamma := e''_i$ and so it must be the case that $\delta_i = \downarrow y$ or $\delta_i = z \gg y$ for some y and z with $y \in \gamma$ and $z \notin \gamma$, or e'_i is not an assignment and then $y \in DG(\pi_i)\Gamma$ for some y in γ. In both cases, $x \in U_0(\pi_j)$, so y interferes with x in π relative to Γ which contradicts the assumed non-interference of Γ.

Now assume likewise that $\rho', \sigma_a \vdash e'^k \Rightarrow v, \sigma$ is not consistent: there is a variable $x_i \in \gamma \in \Gamma$ which is used during evaluation of e'^k, and $\rho'(x_i) \neq \sigma_a(X_\gamma)$. The correct value of $\rho'(x_i)$ is v_i by definition of ρ', and because e'_i must be an assignment $X_\gamma := e''_j$ (for which $\rho, \sigma_{i-1} \vdash e''_i \Rightarrow v_i, \sigma'_i$), we know that $\rho'(x_i) = \sigma_i(X_\gamma)$. So a new value must have been assigned to X_γ after evaluation of e'_i: there must be some e'_j with $j>i$ such that $y \in DG(\pi_j)\Gamma$ or $y \in DG(\delta_j)\Gamma$ for some y belonging to the same variable group γ as x_i. By the assumption made, $\delta_i = \downarrow x_i$ or $\delta_i = z \gg x_i$, and $x_i \in U_0(\pi')$. But then y interferes with x in $\pi = <\pi_1 \delta_1 \ldots \pi_a \delta_a \Diamond \pi'>$ relative to Γ, which contradicts the assumption that Γ is not interfering in pgm.
Hence also $\rho, \sigma_a \vdash e'^k \Rightarrow v, \sigma$ is consistent.

This completes the case analysis for the induction step, and we conclude that every expression evaluation judgement in the evaluation tree is consistent. Since the initial environment ρ_0 was arbitrary, this holds for every possible evaluation of the program. Hence we have proved that the globalization transformation is correct: if Γ is not interfering in pgm, then the transformed program pgm_Γ will produce the same results as pgm. □

This finally allows us to conclude that it is correct to put together the construction of a non-interfering variable grouping and the globalization transformation:

Proposition 3.4-2: (Main correctness proposition for L). Let pgm be an applicative L program, let Γ_{non} be the variable grouping constructed by Algorithm 3.3-1; and let $pgm_{\Gamma non}$ be the program constructed by Algorithm 3.4-1 when applied to pgm and Γ_{non}. Then $pgm_{\Gamma non}$ is at least as strong as pgm.
Proof: Γ_{non} is not interfering in pgm according to Proposition 3.3-1. Therefore by Proposition 3.4-1, $\rho_0 \vdash pgm \Rightarrow v$ implies $\rho_0 \vdash pgm_{\Gamma non} \Rightarrow v$. □

3.5 Overview: The Globalization Method for L

We will close Section 3 with a brief summary of the ideas presented in Sections 2 and 3. Section 2 presented the following concepts and topics:

- *globalization*: the idea of replacing local variables by global ones
- an *example language* L together with an operational semantics for L
- *definition-use paths*, and *use*, *definition*, and *copy* actions
- a *path semantics* for L
- *interference* in a definition-use path
- *variable groups* and *variable groupings*.

Section 3 presented:

- *definition-use grammars*
- *construction* of a definition-use grammar for an L program
- *interference analysis* on definition-use grammars
- construction of a *non-interfering variable grouping*, based on the interference analysis
- the *globalization transformation* for L programs.

The relations among these topics can be illustrated by sketching the steps in analysis and transformation of a given L program pgm:

1. For the given program pgm, a definition-use grammar G_{pgm} is constructed as shown in Section 3.1. This construction depends on the path semantics for L, given in Section 2.3.2.
2. A non-interfering variable grouping Γ_{non} for pgm is constructed, based on the du-grammar G_{pgm}, using the interference analysis ia developed in Section 3.2.
3. The given program pgm is transformed into $pgm_{\Gamma_{non}}$, using the variable grouping Γ_{non} constructed in step 2.

This overview concludes the treatment of globalization for the first order example language L. In the next section, it will be redone (more succinctly) for the higher order example language H which is an extension of L. The entire approach works equally well for the higher order case as we shall see; the only major complication is that a simple flow analysis of the program has to be done to assist the du-grammar construction.

Naturally, H must be given a new syntax, semantics and path semantics, and the grammar construction as well as the transformation in step 3 must be modified to work for H programs. But the interference concept and hence the interference analysis need not be modified at all (on the condition that the path semantics for H is reasonable); neither does the construction of a non-interfering variable grouping, for it relies only on the du-grammar, not on the program. This reflects the strong modularity of the approach to globalization taken in this report.

4. EXTENSION TO HIGHER ORDER FUNCTIONS

This section does part of the work of Sections 2 and 3 once again, this time for a *higher order* example language H. The main new complication is that a flow analysis is needed for the construction of definition-use grammars.

The entire globalization method for H is a modification of that for L. The concepts of path and interference are unchanged, and so the interference analysis on grammars and the construction of non-interfering variable groupings need no modification. The path semantics (Section 4.2) for H is different, however, so the grammar construction must be modified (Sections 4.3 and 4.4). Also, the globalization transformation itself must be changed to suit the syntax of H (Section 4.5). The overall structure of the globalization method is as for L and is recapitulated in Section 4.6.

First the language H which extends L is defined: its syntax, operational semantics and path semantics are given. Then the problems with the grammar construction are explained and a flow analysis is devised to overcome these. For every expression the flow analysis computes (an approximation to) the set of its possible closure values. This is called a *closure analysis*. The definition-use grammar construction is then given, followed by the globalization transformation for H programs and an overview of the globalization method.

4.1 A Higher Order Example Language H

A program in the higher order language H is a system of recursion equations with strict application. The language is designed to be as simple as possible while still being able to model the purely functional parts of current eager functional programming languages such as Standard ML and Scheme, in addition to a strict version of the lambda calculus as used in denotational semantics definitions [Schmidt 1986b].

An H program is a list of global definitions of strict functions or named combinators; the language does not embody the concept of anonymous functions (lambda abstractions) or local function definitions. Therefore local and anonymous definitions in, for example, the lambda calculus need to be transformed into global definitions. Fortunately, this is always possible by the process called *lambda lifting* in [Johnsson 1985] and discussed also in [Hughes 1982] and [Peyton Jones 1987]. In essence, lambda lifting makes an anonymous or local definition global by inventing a unique name for it and making a global definition with that name. Furthermore, it introduces a parameter for each free variable of the anonymous or local definition.

The higher order nature of H is due to the fact that all defined functions are curried. This implies that *partial applications* are legal expressions. When `f` is a function with two parameters, `x` and `y`, it is legal to apply `f` to only one argument expression `e`, and this application is written with the infix application operator `@` as `f@e`. If `e` evaluates to v_x, then the value of the partial application is a *closure*, written `f`(v_x). This closure represents a new function which when given a value v_y for the second parameter of f will produce the same result as would f when applied to both v_x and v_y at once. Such an application, in which values for all the parameters are available, will be called a *full application.* H is a higher order language because a closure value may be passed as a parameter or returned as the result of evaluation. (An aside: *partial evaluation* is partial application plus optimizing syntactic transformations).

In a first order language such as L, non-interference of a variable guarantees that there exists at most one value for it at any time. In a higher order language, this is not true. Even though a variable is not interfering, there may well exist several values for it at the same time. Namely, the values may be *enclosed*, that is, made part of a closure (which represents a functional object) and allocated on the heap. We say that variable x^i_j of function f^i is *enclosed* during an evaluation if a closure for f^i is built that contains a value for variable x^i_j, that is, for parameter position j of f^i.

Non-interference of variable x^i_j of f^i implies that x^i_j need not be made part of each new local environment built for evaluation of f^i's body e^i. Instead the corresponding global variable X should be assigned a new value at the time the environment is built. Non-interference does not imply that x^i_j is never enclosed during an evaluation, however. Non-interference ***and*** non-enclosure of x^i_j imply that at any time there is at most one value for x^i_j. This property is called *single-threading* in [Schmidt 1985]. Non-enclosure of variable x^i_j is rather easy to check for in the simple version of H we use here: x^i_j must be the last variable in the parameter list of f^i.

4.1.1 Syntax and Semantics of Applicative H Programs

A program is a finite set of mutually recursive function definitions, indexed by a finite set I, together with an initial expression e^0:

$$\texttt{def}\ \{f^i\ @\ (x^i_1,\ldots,x^i_{\text{arity}(f^i)}) = e^i\}_{i\in I}\ \texttt{in}\ e^0$$

Every defined function f^i has a fixed arity, written arity(f^i), which must be positive: parameterless functions are not allowed. The language has lexical scope and so only the variables $x^i_1,\ldots, x^i_{\text{arity}(f^i)}$ of f^i can appear in the body e^i. All variables in the program must be distinct. Initial input is through the input parameters of the program, namely those parameters that appear in e^0. Only basic values (booleans, integers, and so on) can be input to a program; closures are not allowed as input or output values. Expressions follow the abstract syntax

e	::=	x	- variable (or, parameter)
	|	f	- defined function symbol
	|	$A(e_1,\ldots,e_a)$	- call of basic function
	|	$e_1 \rightarrow e_2, e_3$	- conditional expression
	|	$e_0\ @\ e_1$	- application

We assume that a set of basic functions A is given, each of which has a fixed non-negative arity arity(A). Zero-arity basic functions are considered constants, and a call `A()` to such a function is just written A. Evaluation of expressions takes place in an environment that binds values to variables. A variable (or parameter) x evaluates to the value it is currently bound to. A function symbol f evaluates to an empty closure which we shall denote by f(). A basic function call $A(e_1,...,e_a)$ is evaluated by evaluating all its argument expressions $e_1,...,e_a$ from left to right and then applying the function to their values. The number of argument expressions must equal arity(A): partial application of basic functions is not allowed. A basic function is only allowed to return those closure values it was given as arguments and cannot evaluate (apply) any closure given to it as an argument; that is, there cannot exist a basic function "apply" that applies a closure. The conditional is as for the L language. An application e_0 @ e_1 is evaluated by evaluating e_0 to obtain a closure $f(v_1, ..., v_m)$ where f is a defined function symbol and $v_1, ..., v_m$ are values for the first m variables of f. It must hold that m < arity(f). Then e_1 is evaluated to obtain a value v_{m+1} for the (m+1)th parameter of f. In case m+1 < arity(f), the application is *partial*, and the result is a new closure $f(v_1, ..., v_{m+1})$. In case m+1 = arity(f), a *full* application is done: a new environment is built which binds the variables of f to $v_1, ..., v_{m+1}$ and the body of f is evaluated in this environment to obtain the value of the application.

Along with the abstract syntax we shall use a more readable concrete one. We will write function definitions without parentheses:

$$f^i\ x^i_1\ \ldots\ x^i_{arity(f^i)} = e^i$$

and we write application e_0 @ e_1 without the application operator, that is, by juxtaposition $e_0\ e_1$. The following disambiguation rules are adopted. Application associates to the left. Application binds more strongly than the conditional, so $e_1 \rightarrow e_2, e_3\ e_4$ means $e_1 \rightarrow e_2, (e_3\ e_4)$. Infix basic functions are allowed and bind less strongly than application but more strongly than conditionals.

Example 4.1.1-1: An interpreter written in H (concrete syntax) for a small imperative language. This is a lambda-lifted version of the example language definition given in Figure 3 (page 301) of [Schmidt 1985]. That example is a denotational semantics definition based on a call by value version of the lambda calculus. The interpreter uses the special basic functions `update` and `access` that work on the store data structure in the interpreter and a special constant `empty` which represents an empty store.

```
def  Cmd c =
        c= "id:=e"                 → Upd id e,
        c= "c1;c2"                 → Compose (Cmd c2) (Cmd c1),
        c= "if e then c1 else c2"  → If e c1 c2,
        c= "repeat c1 until e"     → Compose (F e c1) (Cmd c1),
        ...
     Exp e =
        e= "e1+e2"                 → Add e1 e2,
        e= "id"                    → Look id,
        e= "n"                     → n,
        ...
     Upd idu eu su     = update(idu,Exp eu su,su)
     If ei ci1 ci2 si  = Exp ei si = 0 → Cmd ci1 si, Cmd ci2 si
     F ef cf sf        = Exp ef sf ≠ 0 → F ef cf (Cmd cf sf), sf
     Compose f1 f2 x   = f1 (f2 x)
     Add ea1 ea2 sa    = plus(Exp ea1 sa, Exp ea2 sa)
     Look id sl        = access(id,sl)
in   Cmd c empty                                                        □
```

An operational semantics for H similar to the one for L in Section 2.2.2 is given below, but first we must define some structures to support the description of the semantics.

	BValue = Boolean $\cup$ Integer $\cup$ ...	- basic values,
v:	Value = BValue $\cup$ Closure	- value,
	Closure = $\{ f^i(v_1,...,v_m) \mid v_j \in$ Value, $j=1,..,m;\ 0 \le m < arity(f^i),\ i \in I \}$	
	Input = FreeVars(e^0)	- input parameters of the program,
x:	Var	- local variable of defined function,
ρ:	Env = Var $- \!\!\rightarrow$ Value	- local environment.

The set Input of input parameters must be disjoint from Var. Let the H program

$$\text{pgm} = \ \texttt{def}\ \{f^i\ @\ (x^i_1,...,x^i_{arity(f^i)}) = e^i\}_{i \in I}\ \texttt{in}\ e^0$$

be given. It is evaluated in an initial environment ρ_0: Input $\rightarrow$ BValue, and the result of this is the result of evaluating the initial expression in environment ρ_0:

$$\text{(HP1)} \quad \frac{\rho_0 \vdash e^0 \Rightarrow v}{\rho_0 \vdash \texttt{def}\ \{f^i @\ (x^i_1,...,x^i_{arity(f^i)}) = e^i\}_{i \in I}\ \texttt{in}\ e^0 \Rightarrow v}$$

The judgement below the line concerns evaluation of an H program; the one above the line concerns evaluation of an H expression. The rules for evaluation of expressions are:

(HE1) $\rho \vdash \mathtt{x} \Rightarrow \mathtt{v}$ where $\rho(\mathtt{x}) = \mathtt{v}$

(HE2) $\rho \vdash \mathtt{f}^i \Rightarrow \mathtt{f}^i()$

(HE3) $$\frac{\rho \vdash \mathtt{e}_j \Rightarrow \mathtt{v}_j \quad \text{for } j=1,\ldots,a}{\rho \vdash \mathtt{A}(\mathtt{e}_1,\ldots,\mathtt{e}_a) \Rightarrow \mathtt{v}}$$ where $a = \text{arity}(\mathtt{A})$ and $\mathtt{v} = \mathtt{A}(\mathtt{v}_1,\ldots,\mathtt{v}_a)$

(HE4) $$\frac{\rho \vdash \mathtt{e}_1 \Rightarrow \text{true} \qquad \rho \vdash \mathtt{e}_2 \Rightarrow \mathtt{v}}{\rho \vdash \mathtt{e}_1 \rightarrow \mathtt{e}_2,\mathtt{e}_3 \Rightarrow \mathtt{v}}$$

(HE5) $$\frac{\rho \vdash \mathtt{e}_1 \Rightarrow \text{false} \qquad \rho \vdash \mathtt{e}_3 \Rightarrow \mathtt{v}}{\rho \vdash \mathtt{e}_1 \rightarrow \mathtt{e}_2,\mathtt{e}_3 \Rightarrow \mathtt{v}}$$

(HE6) $$\frac{\rho \vdash \mathtt{e}_0 \Rightarrow \mathtt{f}^i(\mathtt{v}_1,\ldots,\mathtt{v}_m) \qquad \rho \vdash \mathtt{e}_1 \Rightarrow \mathtt{v}_{m+1}}{\rho \vdash \mathtt{e}_0 \mathbin{@} \mathtt{e}_1 \Rightarrow \mathtt{f}^i(\mathtt{v}_1,\ldots,\mathtt{v}_{m+1})}$$ where $m+1 < \text{arity}(\mathtt{f}^i)$

(HE7) $$\frac{\rho \vdash \mathtt{e}_0 \Rightarrow \mathtt{f}^i(\mathtt{v}_1,\ldots,\mathtt{v}_m) \qquad \rho \vdash \mathtt{e}_1 \Rightarrow \mathtt{v}_{m+1} \qquad \rho' \vdash \mathtt{e}^i \Rightarrow \mathtt{v}}{\rho \vdash \mathtt{e}_0 \mathbin{@} \mathtt{e}_1 \Rightarrow \mathtt{v}}$$ where $m+1 = a = \text{arity}(\mathtt{f}^i)$, $\mathtt{e}^i$ is the body of $\mathtt{f}^i$, $\rho' = [\mathtt{x}^i_1 \mapsto \mathtt{v}_1,\ldots,\mathtt{x}^i_a \mapsto \mathtt{v}_a]$.

Notice that the semantics is *deterministic*: if $\rho \vdash \mathtt{e} \Rightarrow \mathtt{v}$ and $\rho \vdash \mathtt{e} \Rightarrow \mathtt{v}'$, then $\mathtt{v}=\mathtt{v}'$.

The version of H presented here allows single applications $\mathtt{e}_0 \mathbin{@} \mathtt{e}_1$ only. As a consequence, no less than n closures will be built during evaluation of a full application $\mathtt{f} \mathbin{@} \mathtt{e}_1 \mathbin{@} \mathtt{e}_2 \mathbin{@} \ldots \mathbin{@} \mathtt{e}_n$, which is undesirable. Another unfortunate effect is that all variables except the last one in the parameter list of a function will be enclosed; that is, they will be made part of a closure and hence allocated on the heap. Therefore only the last variable is really an interesting candidate for globalization.

As a remedy, we might equip H with *multi-applications* of the form $\mathtt{e}_0 \mathbin{@} (\mathtt{e}_1,\ldots,\mathtt{e}_n)$ so that the application of $\mathtt{f}$ above could be written $\mathtt{f} \mathbin{@} (\mathtt{e}_1,\ldots,\mathtt{e}_n)$. Evaluation of this expression should build only the trivial closure $\mathtt{f}()$ and avoid enclosure of the values of $\mathtt{e}_1,\ldots,\mathtt{e}_n$. We do not elaborate on this possibility here; a brief discussion is given in Section 8.1.2.

We keep the single application version of H since it is adequate to demonstrate that our globalization method works for strict higher order languages.

4.1.2 The Imperative Part of H

To get an imperative version of H, we extend the language with the concepts of global variable $\texttt{X} \in$ GloVar and global state σ. Global variables will be written in upper case as for the L language.

How can we add assignments to H? First, we cannot use the syntax `[X:=e]` in call argument expressions as in L. For in a hypothetical application e_0 @ `[X:=e]`, the function expression e_0 may evaluate to several different closures, some for functions that have no globalizable variables and some for functions that have globalizable variables. We want to globalize certain variables, that is, certain (formal) parameter positions. The action taken at evaluation of e_0 @ e_1 must depend on which closure e_0 evaluates to; it cannot depend on the call syntax alone. Thus we shall let the called function's definition decide what happens. This is done by putting a global variable in the parameter list of the function definition. If function `f` is defined by `f X y` = ... where `X` is a global variable and `y` a local one, a call to `f` of form `f 5 7` will assign 5 to the global variable `X` and build a new environment in which `y` is bound to 7.

The question remains *when* the assignment to `X` should be done: at the time of closure building (at evaluation of `f 5`) or only later on, at full application. With the former choice, few variables are likely to be globalizable, for it requires (among other things) that a variable is enclosed in at most one closure at a time. The latter choice will allow more variables to be globalized and is more easily reflected in a path semantics. We chose the latter alternative, assignment at full application, which is adopted also in [Schmidt 1985]. So in the partial application `f 5`, a closure `f`(5) is built, containing the value 5 which will be assigned to `X` only later, in case `f` is fully applied.

Thus the (abstract) syntax for a function definition now is

$$f^i \text{ @ } (\xi^i_1,\ldots,\xi^i_a) = e^i$$

where each ξ^i_j is either a local variable (parameter) or a global variable. All the local variables among $\xi^i_1,\ldots,\xi^i_a$ must be distinct, but the global variables among them need not be. Use of a global variable is now an admissible expression, and the expression syntax is extended with:

$$e \quad ::= \quad X \qquad\qquad \text{- global variable}$$

Evaluation of a global variable `X` returns the value of `X` in the current global state. Evaluation of other expressions except applications proceeds as for applicative H. An application e_0 @ e_1 is evaluated by evaluating e_0 to obtain a closure $f^i(v_1,\ldots,v_m)$. If the application is partial ($m+1 < \text{arity}(f^i)$), then e_1 is evaluated to obtain a value v_{m+1}, and a new closure $f^i(v_1,\ldots,v_m,v_{m+1})$ is returned. If the application is full, that is, $m+1=\text{arity}(f^i)$, then first the global state is updated: if the ξ^i_j in parameter position j of f^i is a global variable `X`, then the corresponding value v_j is assigned to `X`. This is done from left to right for $j=1,\ldots,m$. Then the argument expression e_1 is evaluated in the resulting state to obtain the value v_{m+1}. If ξ^i_{m+1} is a global variable, then its state is updated with v_{m+1}. Finally a new environment ρ' is built using those argument values among

$v_1,\dots,v_m,v_{m+1}$ that were not assigned to global variables, and the body of $\texttt{f}^i$ is evaluated in this new environment.

The formal semantics of H with imperative extensions is expressed using operational semantics as before. We assume that for any given program a finite set Var of local variables and a finite set GloVar of global variables are given and that they are disjoint:

x: Var	- local variable of a defined function,
X: GloVar	- global variable,
ξ: Var $\cup$ GloVar	
σ: State = GloVar $- \to$ Value	- state of global variables,
$\sigma_{init} = \varnothing$: State	- the initial empty state.

The set of input parameters Input = FreeVars(e^0) must be disjoint from the set Var $\cup$ GloVar of local and global variables, and all local variables in the program must be distinct. Moreover, for every function definition $\texttt{f}^i$ @ $(\xi^i_1,\dots,\xi^i_a)$ = e^i, each local variable that occurs in the body e^i must appear in the parameter list $\xi^i_1,\dots,\xi^i_a$.

We introduce some notational devices to simplify environment definitions and state modifications in the semantics. When $\rho \in$ Env, $v \in$ Value, and $\xi \in$ Var $\cup$ GloVar, we take

$\rho[\xi \mapsto v]$ to mean $\rho[x \mapsto v]$ if ξ is a local variable $\xi = x \in$ Var,
ρ if ξ is a global variable.

Consequently, with $v_1, \dots, v_a \in$ Value, and $\xi_1,\dots, \xi_a \in$ Var $\cup$ GloVar,

$\rho = [\xi_j \mapsto v_j$ for $j=1,..,a]$ means $\rho = [x_{b1} \mapsto v_{b1},\dots, x_{bk} \mapsto v_{bk}]$

where $x_{b1},\dots, x_{bk}$ is the subsequence of $\xi_1,\dots, \xi_a$ for which $x_{bj} = \xi_{bj}$ is a local variable. Similarly, for $\sigma \in$ State, we take

$\sigma[\xi \mapsto v]$ to mean $\sigma[X \mapsto v]$ if ξ is a global variable $\xi = X \in$ GloVar,
σ if ξ is a local variable.

Consequently, $\sigma[\xi_j \mapsto v_j$ for $j=1,..,a]$ means $\sigma[X_{g1} \mapsto v_{g1}] \dots [X_{gh} \mapsto v_{gh}]$ where $X_{g1},\dots, X_{gh}$ is the subsequence of $\xi_1,\dots, \xi_a$ for which $X_{gj} = \xi_{gj}$ is a global variable.

With these conventions, we can quite compactly write the new environment ρ' and the new state σ' built by a full application e_0 @ e_1 where e_0 evaluates to the closure $\texttt{f}^i(v_1,\dots,v_m)$ for the defined function $\texttt{f}^i$ @ $(\xi^i_1,\dots,\xi^i_a)$ = e^i and e_1 evaluates to v_{m+1}. The new environment will be $\rho' = [\xi^i_j \mapsto v_j$ for $j=1,..,m+1]$ and the new state will be $\sigma' = \sigma[\xi^i_{m+1} \mapsto v_{m+1}]$ where σ is the state just after evaluation of e_1.

For a given imperative H program

pgm = $\texttt{def}\ \{\texttt{f}^i\ @\ (\xi^i_1,\dots,\xi^i_{arity(\texttt{f}^i)}) = e^i\}_{i \in I}\ \texttt{in}\ e^0$

with Input = FreeVars(e^0), and a given initial environment ρ_0: Input $\to$ BValue, the program evaluation rule is

$$\text{(HIP1)}\quad \frac{\rho_0, \sigma_{\text{init}} \vdash e^0 \Rightarrow v, \sigma}{\rho_0 \vdash \texttt{def}\ \{f^i \,@\, (\xi^i_1,\ldots,\xi^i_{\text{arity}(f^i)}) = e^i\}_{i\in I}\ \texttt{in}\ e^0 \Rightarrow v}$$

The expression evaluation rules are:

$$\text{(HIE1)}\quad \rho, \sigma \vdash x \Rightarrow v, \sigma \qquad \text{where } x \in \text{Var and } \rho(x) = v$$

$$\text{(HIE2)}\quad \rho, \sigma \vdash X \Rightarrow v, \sigma \qquad \text{where } X \in \text{GloVar and } \sigma(X) = v$$

$$\text{(HIE3)}\quad \rho, \sigma \vdash f^i \Rightarrow f^i(), \sigma$$

$$\text{(HIE4)}\quad \frac{\rho, \sigma_{j-1} \vdash e_j \Rightarrow v_j, \sigma_j \quad \text{for } j=1,\ldots,a}{\rho, \sigma_0 \vdash A(e_1,\ldots,e_a) \Rightarrow v, \sigma_a} \qquad \begin{array}{l}\text{where } a = \text{arity}(A)\\ \text{and } v = A(v_1,\ldots,v_a)\end{array}$$

$$\text{(HIE5)}\quad \frac{\begin{array}{l}\rho, \sigma_0 \vdash e_1 \Rightarrow \text{true}, \sigma_1\\ \rho, \sigma_1 \vdash e_2 \Rightarrow v, \sigma_2\end{array}}{\rho, \sigma_0 \vdash e_1 \rightarrow e_2, e_3 \Rightarrow v, \sigma_2}$$

$$\text{(HIE6)}\quad \frac{\begin{array}{l}\rho, \sigma_0 \vdash e_1 \Rightarrow \text{false}, \sigma_1\\ \rho, \sigma_1 \vdash e_3 \Rightarrow v, \sigma_2\end{array}}{\rho, \sigma_0 \vdash e_1 \rightarrow e_2, e_3 \Rightarrow v, \sigma_2}$$

$$\text{(HIE7)}\quad \frac{\begin{array}{l}\rho, \sigma_0 \vdash e_0 \Rightarrow f^i(v_1,\ldots,v_m), \sigma'_0\\ \rho, \sigma'_0 \vdash e_1 \Rightarrow v_{m+1}, \sigma_1\end{array}}{\rho, \sigma_0 \vdash e_0 \,@\, e_1 \Rightarrow f^i(v_1,\ldots,v_{m+1}), \sigma_1} \qquad \text{where } m+1 < \text{arity}(f^i)$$

$$\text{(HIE8)}\quad \frac{\begin{array}{l}\rho, \sigma_0 \vdash e_0 \Rightarrow f^i(v_1,\ldots,v_m), \sigma'_0\\ \rho, \sigma''_0 \vdash e_1 \Rightarrow v_{m+1}, \sigma'_1\\ \rho', \sigma''_1 \vdash e^i \Rightarrow v, \sigma\end{array}}{\rho, \sigma_0 \vdash e_0 \,@\, e_1 \Rightarrow v, \sigma} \qquad \begin{array}{l}\text{where } m+1 = a = \text{arity}(f^i),\\ e^i \text{ is the body of } f^i,\\ \sigma''_0 = \sigma'_0[\xi^i_j \mapsto v_j \text{ for } j=1,\ldots,m],\\ \sigma''_1 = \sigma'_1[\xi^i_{m+1} \mapsto v_{m+1}], \text{ and}\\ \rho' = [\xi^i_j \mapsto v_j \text{ for } j=1,..,a]\end{array}$$

With this semantics the description of imperative H is complete. It is deterministic as is the semantics for applicative H, and clearly extends that semantics: on applicative H programs they yield the same result.

4.2 Path Semantics for H

Here we give the path semantics of applicative H programs. The path semantics will reflect our assumptions about the behaviour of assignment in imperative H programs. In particular, the new environment for evaluation of a function body is not built until a full application of a closure is done. This means that variable definitions $\downarrow x$ happen only at full applications.

As for L programs, the path semantics will prescribe the definition-use path for evaluation of program pgm in initial environment ρ_0 for any given pgm and ρ_0 (provided the evaluation terminates). The path semantics will have precisely the same structure as the operational semantics for applicative H from Section 4.1.1, and will use the same definitions (of the set Path and so on) as the L path semantics from Section 2.3.1. The path semantics is deterministic.

Let an applicative H program

$$\text{pgm} = \texttt{def } \{\texttt{f}^i \,\texttt{@}\, (\texttt{x}^i{}_1,\ldots,\texttt{x}^i{}_{\text{arity}(\texttt{f}^i)}) = \texttt{e}^i\}_{i\in I} \texttt{ in } \texttt{e}^0$$

and an initial environment ρ_0: Input $\rightarrow$ BValue be given. Evaluation of pgm yields the path π if and only if evaluation of the initial expression $\texttt{e}^0$ yields path π:

$$\text{(HPP1)}\quad \frac{\rho_0 \vdash \texttt{e}^0 \Rightarrow \text{v}, \pi}{\rho_0 \vdash \texttt{def } \{\texttt{f}^i \,\texttt{@}\, (\texttt{x}^i{}_1,\ldots,\texttt{x}^i{}_{\text{arity}(\texttt{f}^i)}) = \texttt{e}^i\}_{i\in I} \texttt{ in } \texttt{e}^0 \Rightarrow \text{v}, \pi}$$

For the expression evaluation rules, recall that $\varepsilon \in$ Path is the empty path:

$$\text{(HPE1)}\quad \rho \vdash \texttt{x} \Rightarrow \text{v}, \uparrow\texttt{x} \qquad\qquad \text{where } \rho(\texttt{x}) = \text{v}$$

$$\text{(HPE2)}\quad \rho \vdash \texttt{f}^i \Rightarrow \texttt{f}^i(), \varepsilon$$

$$\text{(HPE3)}\quad \frac{\rho \vdash \texttt{e}_j \Rightarrow \text{v}_j, \pi_j \quad \text{for } j=1,\ldots,a}{\rho \vdash \texttt{A}(\texttt{e}_1,\ldots,\texttt{e}_a) \Rightarrow \text{v}, \pi_1\ldots\pi_a} \qquad\qquad \begin{array}{l}\text{where } a = \text{arity}(\texttt{A})\\ \text{and } \text{v} = \texttt{A}(\text{v}_1,\ldots,\text{v}_a)\end{array}$$

$$\text{(HPE4)}\quad \frac{\begin{array}{l}\rho \vdash \texttt{e}_1 \Rightarrow \text{true}, \pi_1\\ \rho \vdash \texttt{e}_2 \Rightarrow \text{v}, \pi_2\end{array}}{\rho \vdash \texttt{e}_1 \rightarrow \texttt{e}_2, \texttt{e}_3 \Rightarrow \text{v}, \pi_1\pi_2}$$

$$\text{(HPE5)}\quad \frac{\begin{array}{l}\rho \vdash \texttt{e}_1 \Rightarrow \text{false}, \pi_1\\ \rho \vdash \texttt{e}_3 \Rightarrow \text{v}, \pi_3\end{array}}{\rho \vdash \texttt{e}_1 \rightarrow \texttt{e}_2, \texttt{e}_3 \Rightarrow \text{v}, \pi_1\pi_3}$$

$$\text{(HPE6)}\quad \frac{\begin{array}{l}\rho \vdash \texttt{e}_0 \Rightarrow \texttt{f}^i(\text{v}_1,\ldots,\text{v}_m), \pi_0\\ \rho \vdash \texttt{e}_1 \Rightarrow \text{v}_{m+1}, \pi_1\end{array}}{\rho \vdash \texttt{e}_0 \,\texttt{@}\, \texttt{e}_1 \Rightarrow \texttt{f}^i(\text{v}_1,\ldots,\text{v}_{m+1}), \pi_0\pi_1} \qquad\qquad \begin{array}{l}\text{where } a = \text{arity}(\texttt{f}^i)\\ \text{and } m+1 < a\end{array}$$

$$\text{(HPE7)}\quad \frac{\begin{array}{l}\rho \vdash \texttt{e}_0 \Rightarrow \texttt{f}^i(\text{v}_1,\ldots,\text{v}_m), \pi_0\\ \rho \vdash \texttt{e}_1 \Rightarrow \text{v}_{m+1}, \pi_1\\ \rho' \vdash \texttt{e}^i \Rightarrow \text{v}, \pi'\end{array}}{\begin{array}{l}\rho \vdash \texttt{e}_0 \,\texttt{@}\, \texttt{e}_1 \Rightarrow \text{v},\\ \qquad\qquad \pi_0{<}{\downarrow}\texttt{x}^i{}_1\ldots{\downarrow}\texttt{x}^i{}_m\pi_1\delta_1 \lozenge \pi'{>}\end{array}} \qquad\qquad \begin{array}{l}\text{where } a = \text{arity}(\texttt{f}^i),\\ m+1 = a,\\ \texttt{e}^i \text{ is the body of } \texttt{f}^i,\\ \rho' = [\texttt{x}^i{}_1 \mapsto \text{v}_1,\ldots,\texttt{x}^i{}_a \mapsto \text{v}_a], \text{ and}\\ \delta_1 = \Delta(\texttt{x}^i{}_{m+1}, \texttt{e}_1)\end{array}$$

The auxiliary function Δ was defined at the end of Section 2.3.2. With this path semantics we can obtain the definition-use path for evaluation of a given program in a given initial environment. The interference criteria given in Section 2.4 can then be applied to check for interference in this path. To get a computable check for interference on the set of all possible paths of the program, we construct a du-grammar G_{pgm} for the program pgm and make an approximate interference analysis on this grammar that works like the interference criteria for a single path. The general grammar construction for H programs is shown in Section 4.4. The interference analysis is the same as for L programs (Section 3.2). Unlike the grammar construction for L programs, the present one requires the support of a flow analysis, to be presented in the next section.

4.3 Closure Analysis

We shall construct definition-use grammars for H programs as we did for L programs. How can we extend the grammar construction of Section 3.1.2 to work for H programs also? The only new problem is the application expression $e_0 \mathtt{@} e_1$. The grammar nonterminal symbol $N_{e0@e1}$ for this expression must be able to derive all paths that evaluation of $e_0 \mathtt{@} e_1$ can yield. But the path for evaluation of $e_0 \mathtt{@} e_1$ will depend on which closure e_0 evaluates to, as can be seen from path semantics rules HPE6 and HPE7. During the grammar construction we cannot *a priori* know this. One possibility then is to take the safe but very conservative view that e_0 may evaluate to *any* closure possible for the program, and make the grammar derive paths for all these possibilities. This will give a too conservative interference analysis, however, and may spoil opportunities to globalize variables.

A better solution is to do a flow analysis of the program (before the grammar construction) that will allow us to find a more accurate approximation $ca(e_0)$ to the set of closures that e_0 may evaluate to. Using the information collected by this analysis we can make the grammar generate fewer unnecessary paths. We call this a *closure analysis*, abbreviate it ca, and describe it below. It must hold that every closure that evaluation of e_0 may return is a member of the set described by $ca(e_0)$.

Recall that a concrete closure has the form $f^i(v_1,...,v_m)$ where the v's are values and $0 \leq m < arity(f^i)$. The closure analysis will work with *abstract closures* of the form (f^i,m). The abstraction of a concrete closure $f^i(v_1,...,v_m)$ is (f^i,m) in which the actual values $v_1,...,v_m$ are disregarded. An abstract closure (f^i,m) represents the set $\{ f^i(v_1,...,v_m) \mid v_j \in Value, j=1,...,m \}$ of concrete closures.

Abstract application of (f^i,m) to some value v yields a new abstract closure $(f^i,m+1)$ if $m+1 < arity(f^i)$. If $m+1 = arity(f^i)$, then the result is the set of abstractions of closures that standard evaluation of e^i may return. (Note that the argument value vis disregarded completely).

The closure analysis works by building two finite functions (tables) cf and cv. The intention is that $cf(f^i)$ shall contain the set of all abstract closures that (abstract) evaluation of e^i can ever return, and $cv(x)$ shall contain the set of all abstract closures that variable x can ever be bound to. Formally, we define the set AClo of abstract closures and the types of cf and cv as follows:

c:	$\mathrm{AClo} = \{\, (\mathtt{f}^i,m) \mid 0 \le m < \mathrm{arity}(\mathtt{f}^i),\ i \in I \,\}$	- abstract closures,
cf:	$\mathrm{CF} = I \to \wp(\mathrm{AClo})$	- abstract closure values for functions,
cv:	$\mathrm{CV} = \mathrm{Var} \to \wp(\mathrm{AClo})$	- abstract closure values for variables.

Here, I is the indexing set for the functions in the given H program pgm. In this section we will identify each local variable $\mathtt{x}^k{}_j$ in Var with the pair (k,j) where $1 \le j \le \mathrm{arity}(\mathtt{f}^k)$ and $k \in I$. This is admissible because all variables are assumed to be distinct. The functions cf and cv are defined recursively in terms of two functions CA and VA. The intention is that $\mathrm{CA}[\![\mathtt{e}]\!]$cf cv is the set of abstract closures that evaluation of e may return under the assumptions that function f can return only closures in cf(f) and that variable x can be bound only to those abstract closures that are in cv(x). Analogously, $\mathrm{VA}[\![\mathtt{e}]\!](k,j)$ cf cv is the set of abstract closures that evaluation of e may bind to variable $\mathtt{x}^k{}_j$ under those assumptions.

Definition 4.3-1: For a given applicative H program pgm the functions CA and VA are defined as follows (where HExpr is the set of applicative H expressions):

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{CA}: \mathrm{HExpr} \to \mathrm{CF} \to \mathrm{CV} \to \wp(\mathrm{AClo})} \\
\mathrm{CA}[\![\mathtt{x}]\!]\mathrm{cf\ cv} & = & \mathrm{cv}(\mathtt{x}) \\
\mathrm{CA}[\![\mathtt{f}^i]\!]\mathrm{cf\ cv} & = & \{\, (\mathtt{f}^i,0) \,\} \\
\mathrm{CA}[\![\mathtt{A}(\mathtt{e}_1,\ldots,\mathtt{e}_a)]\!]\mathrm{cf\ cv} & = & \mathrm{CA}[\![\mathtt{e}_1]\!]\mathrm{cf\ cv} \cup \ldots \cup \mathrm{CA}[\![\mathtt{e}_a]\!]\mathrm{cf\ cv} \\
\mathrm{CA}[\![\mathtt{e}_1 \to \mathtt{e}_2, \mathtt{e}_3]\!]\mathrm{cf\ cv} & = & \mathrm{CA}[\![\mathtt{e}_2]\!]\mathrm{cf\ cv} \cup \mathrm{CA}[\![\mathtt{e}_3]\!]\mathrm{cf\ cv} \\
\mathrm{CA}[\![\mathtt{e}_0\ @\ \mathtt{e}_1]\!]\mathrm{cf\ cv} & = & \bigcup\{\, \mathrm{aapp}(c,\mathrm{cf},\mathrm{cv}) \mid c \in \mathrm{CA}[\![\mathtt{e}_0]\!]\mathrm{cf\ cv} \,\} \\
\quad \text{where } \mathrm{aapp}((\mathtt{f}^i,m),\mathrm{cf},\mathrm{cv}) & = & \underline{\text{if}}\ m+1=\mathrm{arity}(\mathtt{f}^i)\ \underline{\text{then}}\ \mathrm{cf}(\mathtt{f}^i)\ \underline{\text{else}}\ \{\, (\mathtt{f}^i,m+1) \,\}
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathrm{VA}: \mathrm{HExpr} \to \mathrm{Var} \to \mathrm{CF} \to \mathrm{CV} \to \wp(\mathrm{AClo})} \\
\mathrm{VA}[\![\mathtt{x}]\!](k,j)\ \mathrm{cf\ cv} & = & \{\} \\
\mathrm{VA}[\![\mathtt{f}^i]\!](k,j)\ \mathrm{cf\ cv} & = & \{\} \\
\mathrm{VA}[\![\mathtt{A}(\mathtt{e}_1,\ldots,\mathtt{e}_a)]\!](k,j)\ \mathrm{cf\ cv} & = & \mathrm{VA}[\![\mathtt{e}_1]\!](k,j)\ \mathrm{cf\ cv} \cup \ldots \cup \mathrm{VA}[\![\mathtt{e}_a]\!](k,j)\ \mathrm{cf\ cv} \\
\mathrm{VA}[\![\mathtt{e}_1 \to \mathtt{e}_2, \mathtt{e}_3]\!](k,j)\ \mathrm{cf\ cv} & = & \mathrm{VA}[\![\mathtt{e}_1]\!](k,j)\ \mathrm{cf\ cv} \cup \ldots \cup \mathrm{VA}[\![\mathtt{e}_3]\!](k,j)\ \mathrm{cf\ cv} \\
\mathrm{VA}[\![\mathtt{e}_0\ @\ \mathtt{e}_1]\!](k,j)\ \mathrm{cf\ cv} & = & C_{k,j} \cup \mathrm{VA}[\![\mathtt{e}_0]\!](k,j)\ \mathrm{cf\ cv} \cup \mathrm{VA}[\![\mathtt{e}_1]\!](k,j)\ \mathrm{cf\ cv} \\
\quad \text{where } C_{k,j} & = & \mathrm{CA}[\![\mathtt{e}_1]\!]\mathrm{cf\ cv} \quad \text{if } (\mathtt{f}^k,j-1) \in \mathrm{CA}[\![\mathtt{e}_0]\!]\mathrm{cf\ cv} \\
 & = & \{\} \qquad\qquad\qquad \text{otherwise}
\end{array}
$$

□

Clearly for the functions cf and cv we want that $\mathrm{cf}(\mathtt{f}^i)$ equals $\mathrm{CA}[\![\mathtt{e}^i]\!]$cf cv and that $\mathrm{cv}(\mathtt{x}^k{}_j)$ is the union of $\mathrm{VA}[\![\mathtt{e}]\!](k,j)$ cf cv for all expressions e in pgm. At the same time $\mathrm{cf}(\mathtt{f}^i)$ and cv(x) should contain as few elements as possible or else the closure analysis will be unnecessarily imprecise. This indicates that the cf and cv we want are simultaneous least fixed points of $\lambda(\mathrm{cf},\mathrm{cv}).\ \lambda i.\ \mathrm{CA}[\![\mathtt{e}^i]\!]\mathrm{cf\ cv}$ and $\lambda(\mathrm{cf},\mathrm{cv}).\ \lambda(k,j).\ \bigcup\{\mathrm{VA}[\![\mathtt{e}^i]\!](k,j)\ \mathrm{cf\ cv} \mid i \in I\,\}$.

This insight is exploited in

Definition 4.3-2: The approximate *closure analysis* ca: HExp $\rightarrow \wp$(AClo) for a given applicative H program pgm is defined by

$$\text{ca}(e) \;=\; \text{CA}[\![e]\!]\text{cf cv}$$

where (cf,cv) is the pointwise inclusion-least solution to

$$\begin{aligned} \text{cf}(f^i) &= \text{CA}[\![e^i]\!]\text{cf cv} && \text{for } i \in I \\ \text{cv}(x^k{}_j) &= \bigcup\{\, \text{VA}[\![e^i]\!](k,j)\ \text{cf cv} \mid i \in I \,\} && \text{for } x^k{}_j \in \text{Var} \end{aligned}$$ □

The required solution (cf,cv) in the definition exists, is unique and is effectively computable, for λ(cf,cv). λi. CA$[\![e^i]\!]$cf cv and λ(cf,cv). λ(k,j). $\bigcup$ {VA$[\![e^i]\!]$(k,j) cf cv | i $\in$ I } are monotonic on CF $\times$ CV when this set is made into a lattice by the pointwise inclusion ordering on each component. The correctness requirement on this analysis is: if there is a terminating evaluation of program pgm in which evaluation of e returns closure $f^i(v_1,\ldots,v_m)$, then its abstraction (f^i,m) is in ca(e). This is proved to be the case in Proposition 5.1-1.

The construction of cf and cv can be implemented simply by an iterative algorithm that starts with cf = λf^i.{} and cv = λx.{} and repeatedly recomputes cf(f^i) := CA$[\![e^i]\!]$cf cv and cv($x^k{}_j$) := $\bigcup$ {VA$[\![e^i]\!]$(k,j) cf cv | i $\in$ I } until cf and cv stabilize which will happen in finitely many iterations.

Example 4.3-1: For the interpreter program from Example 4.1.1-1, the closure analysis will construct the following functions cf and cv:

cf(Cmd)	= { (Upd,2), (Compose,2), (If,3) }	
cf(Exp)	= { (Add,2), (Look,1) }	
cf(f)	= { }	for all other functions f
cv(f1)	= { (Upd,2), (Compose,2), (If,3), (F,2) }	
cv(f2)	= { (Upd,2), (Comp,2), (If,3) }	
cv(x)	= { }	for all other variables x

With these cf and cv, the closure analysis yields ca(Cmd c) = { (Upd,2), (Compose,2), (If,3) }. Using this fact and cf(Upd) = cf(Compose) = cf(If) = { } it is easy to see that ca(Cmd c empty) = { } which shows that the result of the interpreter cannot be a closure. □

4.4 Definition-Use Grammar Construction for H

For a given applicative H program pgm we can construct a definition-use grammar G_{pgm} that derives every path possible for the program. The construction is quite similar to that for L programs given in Section 3.1.2, except that the present construction needs the results of the approximate closure analysis ca developed in Section 4.3.

As for an L program, the du-grammar for an H program pgm will have start nonterminal N_{pgm}, and the other nonterminals will be of form N_e where e is a (sub)expression in pgm. The nonterminals are also interpreted in the same way: if π is a possible path for program pgm, then π must be derivable from the start nonterminal N_{pgm}. Likewise, if π is a possible path of expression e in some evaluation, then π must be derivable from N_e.

The grammar construction is based on the H path semantics (Section 4.2). The only case in the construction which is not straightforward is application e_0 @ e_1. The closure analysis must tell which (abstract) closures e_0 may return, and partial applications (path semantics rule HPE6) must be distinguished from full applications (rule HPE7).

Algorithm 4.4-1: Definition-use grammar construction for H.
Input: An applicative H program pgm = `def` $\{f^i$ @ $(x^i_1,...,x^i_{arity(f^i)}) = e^i\}_{i\in I}$ `in` e^0.
Output: The set R of rules of definition-use grammar G_{pgm} for pgm.

```
R := { $N_{pgm} \to N_{e^0}$ };
while there is a nonterminal that is used but not defined in R do
    choose such a nonterminal $N_e$;
    case e of
        variable x      : R := R ∪ { $N_x \to \uparrow x$ };
        $f^i$              : R := R ∪ { $N_{f^i} \to \varepsilon$ };
        A(e1,...,ea)    : R := R ∪ { $N_{A(e1,...,ea)} \to N_{e1}N_{e2}...N_{ea}$ };
        e1→e2,e3        : R := R ∪ { $N_{e1\to e2,e3} \to N_{e1}N_{e2}$,
                                     $N_{e1\to e2,e3} \to N_{e1}N_{e3}$ };
        e0 @ e1:
            for each $(f^i,m) \in ca(e0)$ do
                if $m+1 = arity(f^i)$ then
                    R := R ∪ { $N_{e0@e1} \to N_{e0}<\downarrow x^i_1...\downarrow x^i_m N_{e1}\delta_1 \Diamond N_{e^i}>$ }
                    where $\delta_1 = \Delta(x^i_{m+1}, e1)$
                else (* $m+1 < arity(f^i)$ *)
                    R := R ∪ { $N_{e0@e1} \to N_{e0}N_{e1}$ }
                endif
    endcase
endwhile
```
□

This grammar construction algorithm could be improved by suppressing generation of ε-rules and rules that can produce only a single terminal symbol. Notice that the right hand sides of the rules generated have the form required for members of Rhs (defined in Section 3.1.1).

The correctness of the grammar construction heavily depends on the correctness of the closure analysis, but otherwise the correctness proof is completely similar to that given for the L grammar construction in Section 3.1.

Example 4.4-1: A definition-use grammar for the interpreter from Example 4.1.1-1. The construction of this grammar relies on the results of the approximate closure analysis shown in Example 4.3-1. The grammar below has been rewritten to reduce the number of rules.

N_{pgm}	→ <↓c ◊ $N_{\mathtt{Cmd}}$><↓idu↓eu↓su ◊ $N_{\mathtt{Upd}}$>
N_{pgm}	→ <↓c ◊ $N_{\mathtt{Cmd}}$><↓f1↓f2↓x ◊ $N_{\mathtt{Compose}}$>
N_{pgm}	→ <↓c ◊ $N_{\mathtt{Cmd}}$><↓ei↓ci1↓ci2↓si ◊ $N_{\mathtt{If}}$>
$N_{\mathtt{Cmd}}$	→ ↑c↑c↑c
$N_{\mathtt{Cmd}}$	→ ↑c↑c<↑c↓c ◊ $N_{\mathtt{Cmd}}$> <↑c↓c ◊ $N_{\mathtt{Cmd}}$>
$N_{\mathtt{Cmd}}$	→ ↑c↑c↑c↑c↑c↑c
$N_{\mathtt{Cmd}}$	→ ↑c↑c↑c↑c↑c↑c <↑c↓c ◊ $N_{\mathtt{Cmd}}$>
$N_{\mathtt{Exp}}$	→ ↑e↑e↑e
$N_{\mathtt{Exp}}$	→ ↑e↑e↑e
$N_{\mathtt{Exp}}$	→ ↑e↑e↑e↑e
$N_{\mathtt{Upd}}$	→ ↑idu<↑eu eu»e ◊ $N_{\mathtt{Exp}}$> <↓ea1↓ea2↑su su»sa ◊ $N_{\mathtt{Add}}$>↑su
$N_{\mathtt{Upd}}$	→ ↑idu<↑eu eu»e ◊ $N_{\mathtt{Exp}}$> <↓id↑su su»sl ◊ $N_{\mathtt{Look}}$>↑su
$N_{\mathtt{If}}$	→ <↑ei ei»e ◊ $N_{\mathtt{Exp}}$> $N_{\mathtt{E-I}}$ <↑ci1 ci1»c ◊ $N_{\mathtt{Cmd}}$> $N_{\mathtt{C-I}}$
$N_{\mathtt{If}}$	→ <↑ei ei»e ◊ $N_{\mathtt{Exp}}$> $N_{\mathtt{E-I}}$ <↑ci2 ci2»c ◊ $N_{\mathtt{Cmd}}$> $N_{\mathtt{C-I}}$
$N_{\mathtt{E-I}}$	→ <↓ea1↓ea2↑si si»sa ◊ $N_{\mathtt{Add}}$>
$N_{\mathtt{E-I}}$	→ <↓id↑si si»sl ◊ $N_{\mathtt{Look}}$>
$N_{\mathtt{C-I}}$	→ <↓idu↓eu↑si si»su ◊ $N_{\mathtt{Upd}}$>
$N_{\mathtt{C-I}}$	→ <↓f1↓f2↑si si»x ◊ $N_{\mathtt{Compose}}$>
$N_{\mathtt{C-I}}$	→ <↓ei↓ci1↓ci2↑si si»si ◊ $N_{\mathtt{If}}$>
$N_{\mathtt{F}}$	→ <↑ef ef»e ◊ $N_{\mathtt{Exp}}$>$N_{\mathtt{E-F}}$↑ef↑cf<↓ef↓cf $N_{\mathtt{C-F}}$ ↓sf ◊ $N_{\mathtt{F}}$>
$N_{\mathtt{F}}$	→ <↑ef ef»e ◊ $N_{\mathtt{Exp}}$>$N_{\mathtt{E-F}}$↑sf
$N_{\mathtt{E-F}}$	→ <↓ea1↓ea2↑sf sf»sa ◊ $N_{\mathtt{Add}}$>
$N_{\mathtt{E-F}}$	→ <↓id↑sf sf»sl ◊ $N_{\mathtt{Look}}$>
$N_{\mathtt{C-F}}$	→ <↑cf cf»c ◊ $N_{\mathtt{Cmd}}$> <↓idu↓eu↑sf sf»su ◊ $N_{\mathtt{Upd}}$>
$N_{\mathtt{C-F}}$	→ <↑cf cf»c ◊ $N_{\mathtt{Cmd}}$> <↓f1↓f2↑sf sf»x ◊ $N_{\mathtt{Compose}}$>
$N_{\mathtt{C-F}}$	→ <↑cf cf»c ◊ $N_{\mathtt{Cmd}}$> <↓ei↓ci1↓ci2↑sf sf»si ◊ $N_{\mathtt{If}}$>
$N_{\mathtt{Compose}}$	→ <↓ef↓cf $N_{\mathtt{f2}}$ ↓sf ◊ $N_{\mathtt{F}}$>
$N_{\mathtt{Compose}}$	→ <↓idu↓eu $N_{\mathtt{f2}}$ ↓su ◊ $N_{\mathtt{Upd}}$>
$N_{\mathtt{Compose}}$	→ <↓f1↓f2 $N_{\mathtt{f2}}$ ↓x ◊ $N_{\mathtt{Compose}}$>
$N_{\mathtt{Compose}}$	→ <↓ei↓ci1↓ci2 $N_{\mathtt{f2}}$ ↓si ◊ $N_{\mathtt{If}}$>
$N_{\mathtt{f2}}$	→ <↓idu↓eu↑x x»su ◊ $N_{\mathtt{Upd}}$>
$N_{\mathtt{f2}}$	→ <↓f1↓f2↑x x»x ◊ $N_{\mathtt{Compose}}$>
$N_{\mathtt{f2}}$	→ <↓ei↓ci1↓ci2↑x x»si ◊ $N_{\mathtt{If}}$>
$N_{\mathtt{E-A}}$	→ <↓ea1↓ea2↑sa sa»sa ◊ $N_{\mathtt{Add}}$>
$N_{\mathtt{E-A}}$	→ <↓id↑sa sa»sl ◊ $N_{\mathtt{Look}}$>
$N_{\mathtt{Look}}$	→ ↑id↑sl □

From this du-grammar it can be deduced that with γ = {su, si, sf, x, sa, sl} and Γ = {γ}, none of the variables in γ interferes with each other relative to Γ in the interpreter. Hence all variables in γ can be replaced by one global store variable S. Furthermore, every one of these variables appears as the last one in its parameter list and therefore its value is never enclosed. This is the result obtained in the paper [Schmidt 1985] from which the interpreter example originates.

4.5 The Globalization Transformation for H

The transformation that globalizes variables in an applicative H program is quite similar to the transformation for L programs presented in Section 3.4. The presentation here will therefore be fairly terse.

The transformation takes as input an applicative H program pgm and a variable grouping Γ for pgm, and produces as output an imperative H program pgm_Γ. This program uses one global variable called X_γ for each variable group $\gamma \in \Gamma$ and is constructed by transforming each function definition $\texttt{f}^i \texttt{ @ } (x^i_1,\ldots,x^i_{\text{arity}(\texttt{f}^i)}) = e^i$ as follows. Every parameter x^i_j in the parameter list which belongs to some variable group $\gamma \in \Gamma$ is replaced by the corresponding global variable X_γ. Consistently with this, the body expression e^i is transformed so that every use of variable x^i_j is replaced by a use of X_γ.

The effect is that a full application of the transformed function $\texttt{f}^i$ will build an environment ρ' that contains bindings only for those x^i_j that do not belong to any variable group in Γ. The value for a variable that does belong to a variable group γ will be assigned to the corresponding global variable X_γ instead. These assignments to global variables during the building of the new environment ρ' happen strictly in the order of evaluation of the argument expressions, that is, from left to right. If the full application is the result of an application of a non-empty closure $\texttt{f}^i(v_1,\ldots,v_m)$ to one further argument, and $j \le m$, then the value of v_j for x^i_j comes from the closure, but it is assigned to the global variable X_γ only at full application.

Algorithm 4.5-1: The globalization transformation for H programs.
Input: An applicative H program

$$\text{pgm} = \quad \texttt{def } \{\texttt{f}^i \texttt{@ } (x^i_1,\ldots,x^i_{\text{arity}(\texttt{f}^i)}) = e^i\}_{i\in I} \texttt{ in } e^0$$

and a variable grouping Γ for pgm which is not interfering in pgm.
Output: An imperative H program in which every variable group $\gamma \in \Gamma$ is replaced by a global variable X_γ.
The transformed program is

$$\text{pgm}_\Gamma = \quad \texttt{def } \{\texttt{f}^i \texttt{ @ } (\xi^i_1,\ldots,\xi^i_{\text{arity}(\texttt{f}^i)}) = \texttt{te}^i\}_{i\in I} \texttt{ in } e^0$$

where ξ^i_j is the global variable X_γ if x^i_j of pgm belongs to variable group $\gamma \in \Gamma$, and ξ^i_j is the local variable x^i_j of pgm otherwise, $j=1,\ldots,\text{arity}(\texttt{f}^i)$. Each body expression e^i is transformed into $\texttt{te}^i = T[\![e^i]\!]\Gamma$ using the expression transformation T defined below. The initial expression e^0 is not transformed, for it contains no variable x that belongs to a variable group. (It can contain only input parameters of the program and these are not considered variables). □

Definition 4.5-1: The globalization transformation T for applicative H expressions is defined as follows (where HExpr denotes the set of applicative H expressions and HIExpr denotes the set of imperative ones):

$$
\begin{array}{lll}
\multicolumn{3}{l}{T\text{: HExpr} \rightarrow \text{VGrouping} \rightarrow \text{HIExpr}} \\
T[\![\mathtt{x}]\!]\Gamma & = \mathtt{x}_\gamma & \text{if } x \in \gamma \text{ for some } \gamma \in \Gamma \\
 & = \mathtt{x} & \text{otherwise} \\
T[\![\mathtt{f}^i]\!]\Gamma & = \mathtt{f}^i & \\
T[\![\mathtt{A}(e_1,\ldots,e_a)]\!]\Gamma & = \mathtt{A}(T[\![e_1]\!]\Gamma,\ldots, T[\![e_a]\!]\Gamma) & \\
T[\![e_1 \rightarrow e_2, e_3]\!]\Gamma & = T[\![e_1]\!]\Gamma \rightarrow T[\![e_2]\!]\Gamma, T[\![e_3]\!]\Gamma & \\
T[\![e_0 @ e_1]\!]\Gamma & = T[\![e_0]\!]\Gamma \mathbin{@} T[\![e_1]\!]\Gamma &
\end{array}
$$
□

The correctness of the globalization transformation is shown in Proposition 5.2-1.

4.6 Overview: The Globalization Method for H

Above we have presented the modifications to the globalization method necessary for a higher order language H. In summary, the method works in these steps for an applicative H program pgm:

1. Do the closure analysis to obtain the functions cf and cv for pgm (Section 4.3)
2. Use these results to construct a du-grammar G_{pgm} (Section 4.4)
3. Construct a non-interfering variable grouping Γ_{non} by repeated application of the interference analysis to the grammar (Sections 3.2 and 3.3).
4. Transform pgm into $pgm_{\Gamma_{non}}$ in which every variable group in Γ_{non} is replaced by a global variable (Section 4.5).

5. CORRECTNESS FOR H

This section proves the correctness of the globalization method for H. Since the globalization method for H reuses many of the components from L, only the closure analysis from Section 4.3 and the transformation from Section 4.5 will be proved in detail. The steps in the overall correctness proof for H are:

1. The grammar construction is similar to that for L and no new proof will be given. Its correctness depends on the correctness of the closure analysis, however.
2. The closure analysis is proved correct in Section 5.1.
3. The interference analysis is as for L and no new proof will be given.
4. The construction of a non-interfering variable grouping is exactly as for L and no new proof will be given.
5. The globalization transformation is proved correct in Section 5.2.

The first four steps guarantee that a non-interfering variable grouping Γ_{non} is in fact constructed by our methods. The fifth step guarantees that given pgm and a variable grouping Γ not interfering in pgm, the transformed pgm_Γ is at least as strong as pgm.

5.1 Correctness of the Closure Analysis

Let an applicative H program

$$\text{pgm} = \texttt{def } \{\texttt{f}^i \texttt{@} (\texttt{x}^i{}_1, \ldots, \texttt{x}^i{}_{\text{arity}(\texttt{f}^i)}) = \texttt{e}^i\}_{i \in I} \texttt{ in } \texttt{e}^0$$

be given, and let ca: HExpr $\rightarrow \wp$(AClo) represent the results of the closure analysis for pgm.

The correctness requirement for the closure analysis is: if during some evaluation of pgm, expression e evaluates to a closure $\texttt{f}^i(v_1,\ldots,v_m)$, then ca(e) contains the approximation $(\texttt{f}^i,m)$ of this closure. This is a property of the set of possible evaluation trees for pgm, relative to the obtained closure analysis ca. For every (finite) evaluation tree and every expression evaluation judgement $\rho \vdash e \Rightarrow v$ in the tree, if v is a closure $\texttt{f}^i(v_1,\ldots,v_m)$, then $(\texttt{f}^i,m) \in \text{ca}(e)$.

This will be proved by induction on the structure of evaluation trees for pgm. In outline, we will prove that if ca safely describes the closures in ρ, then it safely describes the result v of evaluation of e in $\rho \vdash e \Rightarrow v$. This will suffice, for an initial environment ρ_0: Input $\rightarrow$ BValue cannot contain any closures at all and hence trivially is safely described by ca. But then so is every judgement in an evaluation tree, and from this it follows that its results are safely described by ca.

First we must formalize the meaning of "safely describes".

Definition 5.1-1: Closure $\texttt{f}^i(v_1,\ldots,v_m)$ is *c-safe for* ca iff $v_j = \texttt{f}'(v'_1,\ldots,v'_{m'}) \in$ Closure implies that $(\texttt{f}',m') \in \text{ca}(\texttt{x}^i{}_j)$ and v_j is itself c-safe for ca, j=1,...,m. □

Intuitively, $\mathtt{f}^i(v_1,...,v_m)$ is c-safe for ca if v_j being a closure implies that the abstraction of v_j is a possible value for parameter position j of $\mathtt{f}^i$ according to the closure analysis.

Definition 5.1-2: Let $\rho \vdash e \Rightarrow v$ be an expression evaluation judgement. The judgement is *ρ-safe for* ca iff $\rho(x) = \mathtt{f}^i(v_1,...,v_m) \in$ Closure implies that $(\mathtt{f}^i,m) \in$ ca(x) and $\mathtt{f}^i(v_1,...,v_m)$ is c-safe. The judgement is *v-safe for* ca iff $v = \mathtt{f}^i(v_1,...,v_m) \in$ Closure implies that $(\mathtt{f}^i,m) \in$ ca(e) and v is c-safe. □

Intuitively, $\rho \vdash e \Rightarrow v$ is ρ-safe if all the closures in ρ (and the closures these contain and so on) are possible values (according to the closure analysis) for the parameter positions they occupy. The judgement is v-safe if v is a possible value of e according to ca whenever v is a closure.

Proposition 5.1-1: Let ρ_0: Input → BValue be an initial environment. Every expression evaluation judgement in the (finite) evaluation tree for $\rho_0 \vdash e^0 \Rightarrow v$ is v-safe for ca.
Proof: Since this root judgement is trivially ρ-safe for ca, every judgement in the tree is v-safe by Lemma 5.1-1, and so in particular the root judgement is v-safe. □

Lemma 5.1-1: Let a (sub)expression e of pgm be given and assume the evaluation tree for $\rho \vdash e \Rightarrow v$ is finite. If this (root) judgement is ρ-safe, then every judgement in the evaluation tree is v-safe for ca.
Proof: By induction on the structure of the evaluation tree for $\rho \vdash e \Rightarrow v$. The proof depends on the operational semantics for H given in Section 4.1.1.

Case $e \equiv x$ matches rule HE1. We must have $v = \rho(x)$. By ρ-safety, $v = \mathtt{f}^i(v_1,...,v_m) \in$ Closure implies that $(\mathtt{f}^i,m) \in$ ca(x) and that $\mathtt{f}^i(v_1,...,v_m)$ is c-safe. Therefore the (only) judgement in the tree is v-safe.

Case $e \equiv \mathtt{f}^i$ matches rule HE2. Clearly, $v = \mathtt{f}^i() \in \mathrm{ca}(\mathtt{f}^i) = \{ (\mathtt{f}^i,0) \}$. Thus the (only) judgement in the tree is v-safe.

Case $e \equiv \mathtt{A}(e_1,...,e_a)$ matches rule HE3 with a = arity(A). Every premise judgement $\rho \vdash e_j \Rightarrow v_j$ for j=1,..,a is ρ-safe, and hence by the induction hypothesis, all judgements in their evaluation trees are v-safe. It remains to prove the v-safety of the root judgement $\rho \vdash e \Rightarrow v$. If $v = \mathtt{f}'(v'_1,...,v'_{m'})$, then $v = v_j$ for some $j \in \{1,...,a\}$, for the basic function A cannot return any closure not passed to it as an argument. Therefore $(\mathtt{f}',m') \in \mathrm{ca}(e_j)$ by the v-safety of $\rho \vdash e_j \Rightarrow v_j$ (which also proves that v is c-safe). Hence $(\mathtt{f}',m') \in \mathrm{ca}(e) = \mathrm{ca}(e_1) \cup ... \cup \mathrm{ca}(e_a)$, which proves the v-safety of the root judgement.

Case $e \equiv e_1 \rightarrow e_2, e_3$ matches rule HE4 or HE5. Assume $\rho \vdash e_1 \Rightarrow$ true so rule HE4 matches; the other case is similar. Clearly, $\rho \vdash e_j \Rightarrow v_j$ for j=1,2 are both ρ-safe, so all the judgements in their evaluation trees are v-safe. It remains to prove the v-safety of the root judgement. But if $v = \mathtt{f}'(v'_1,...,v'_{m'})$, then $(\mathtt{f}',m') \in \mathrm{ca}(e_2)$ by the v-safety of $\rho \vdash e_2 \Rightarrow v$, and so $(\mathtt{f}',m') \in \mathrm{ca}(e)$. Also, v is c-safe, and hence the root judgement is v-safe.

Case $e \equiv e_0 @ e_1$ matches rule HE6 or HE7. First assume $\rho \vdash e_0 \Rightarrow f^i(v_1,...,v_m)$ with $m+1 < \text{arity}(f^i)$ so rule HE6 applies. Both premise judgements are ρ-safe, and so by the induction hypothesis all judgements in their evaluation trees are v-safe. It remains to be shown that the root judgement is v-safe. Its value must be $v = f^i(v_1,...,v_m,v_{m+1})$ for some v_{m+1}. But by the v-safety of $\rho \vdash e_0 \Rightarrow f^i(v_1,...,v_m)$, it follows that $(f^i,m) \in ca(e_0)$ and so $(f^i,m+1) \in ca(e_0 @ e_1)$. The c-safety of v follows because $f^i(v_1,...,v_m)$ and v_{m+1} are c-safe due to the v-safety of the premise judgements, and because if $v_{m+1} = f'(v'_1,...,v'_{m'})$, then $(f',m') \in ca(e_1) \subseteq VA[\![e_0 @ e_1]\!](i,m+1) cf\ cv \subseteq cv(x^i_{m+1}) = ca(x^i_{m+1})$ which is a consequence of v-safety of the premises and e being a (sub)expression in pgm. Thus also the root judgement is v-safe.

Now to the second case: $m+1 = a = \text{arity}(f^i)$ which matches rule HE7. Both $\rho \vdash e_0 \Rightarrow f^i(v_1,...,v_m)$ and $\rho \vdash e_1 \Rightarrow v_{m+1}$ are ρ-safe, and hence all judgements in their evaluation trees are v-safe by the induction hypothesis. First we prove ρ-safety of $\rho' \vdash e^i \Rightarrow v$ where $\rho' = [x^i_1 \mapsto v_1,...,x^i_a \mapsto v_a]$ and e^i is the body of f^i. But ρ-safety of $\rho \vdash e_0 \Rightarrow f^i(v_1,...,v_m)$ implies c-safety of $f^i(v_1,...,v_m)$. Moreover, $v_{m+1} = f'(v'_1,...,v'_{m'}) \in$ Closure implies $(f',m') \in ca(e_1) \subseteq VA[\![e_0 @ e_1]\!](i,m+1) cf\ cv \subseteq cv(x^i_{m+1}) = ca(x^i_{m+1})$ by the v-safety of $\rho \vdash e_1 \Rightarrow v_{m+1}$, by definition of ca, and by e being a (sub)expression of pgm. This shows that $\rho' \vdash e^i \Rightarrow v$ is ρ-safe, and hence by the induction hypothesis, every judgement in its evaluation tree is v-safe. In particular, if $v = f'(v'_1,...,v'_{m'}) \in$ Closure, then v is c-safe and $(f',m') \in ca(e^i)$ which proves the v-safety of the root judgement, for then $(f',m') \in ca(e_0 @ e_1) = cf(f^i) = ca(e^i)$. Hence every judgement is v-safe.

This completes the proof by induction on the structure of the evaluation trees. Hence if an evaluation tree for e has a ρ-safe root, then every judgement in the tree is v-safe. □

Thus the closure analysis is correct in the sense mentioned above. In the next section we prove that the globalization transformation and the entire method are correct.

5.2 Correctness of the Globalization Transformation for H

The globalization transformation for H programs presented in Section 4.5 is different from that for L programs and requires a new proof.

Let an applicative H program pgm be given and let Γ be a variable grouping not interfering in pgm. Then the transformed program pgm_Γ is at least as strong as pgm: if $\rho_0 \vdash pgm \Rightarrow v$ then $\rho_0 \vdash pgm_\Gamma \Rightarrow v$.

We will prove that for every (sub)expression e in pgm and environment ρ,

if
- $\rho \vdash e \Rightarrow v$,
- Γ is not interfering in the evaluation of e,
- the environment ρ_Γ and the state σ are consistent with ρ, and
- $e' = T[\![e]\!]\Gamma$ is the transformed version of e,

then
- $\rho_\Gamma, \sigma \vdash e' \Rightarrow v, \sigma''$ for some σ''.

That is, the transformed expression e' evaluates to v also. Consistency of ρ_Γ and σ with ρ means that those variables in dom(ρ) which are not globalized must have the same value in ρ_Γ as in ρ, and those that are globalized and are used in e must have the same value in σ as in ρ. By the variables used in e we mean those in $U_0(\pi)$ where $\rho \vdash e \Rightarrow v, \pi$ by the path semantics. The concept of consistency is made more precise in the definition below.

Definition 5.2-1: The tuple (ρ_Γ, σ, e) *is consistent with* ρ iff

1. If $x \in \text{dom}(\rho)$ and $x \notin \bigcup\Gamma$,
   then $x \in \text{dom}(\rho_\Gamma)$ and $\rho_\Gamma(x) = \rho(x)$.
2. If $x \in \text{dom}(\rho)$ and $x \in \gamma \in \Gamma$ and x is referenced during evaluation of e,
   then $x_\gamma \in \text{dom}(\sigma)$ and $\sigma(x_\gamma) = \rho(x)$. □

Variable x is referenced during evaluation of e precisely if $x \in U_0(\pi)$ where $\rho \vdash e \Rightarrow v, \pi$. Notice that if (ρ_Γ, σ, e) is consistent with ρ, then so is $(\rho_\Gamma, \sigma, e'')$ for every e" which is a subexpression of e.

Proposition 5.2-1: Let pgm be an applicative H program pgm and assume that Γ is not interfering in pgm. For every initial environment ρ_0: Input $\rightarrow$ BValue, if $\rho_0 \vdash \text{pgm} \Rightarrow v$, then
$\rho_0 \vdash \text{pgm}_\Gamma \Rightarrow v$ where pgm_Γ is the result of applying Algorithm 4.5-1 to pgm and Γ.
Proof: Γ is not interfering in path π where $\rho_0 \vdash e^0 \Rightarrow v, \pi$ in the H path semantics, and $(\rho_0, \sigma_{init}, e^0)$ is trivially consistent with ρ_0. Hence $\rho_0, \sigma_{init} \vdash e^0 \Rightarrow v, \sigma$ in pgm_Γ for some σ by Lemma 5.2-1. □

Lemma 5.2-1: Let e be a (sub)expression of pgm, let ρ be an environment, and let Γ be a variable grouping for pgm. Assume that

- $\rho \vdash e \Rightarrow v, \pi$ in the H path semantics,
- Γ is not interfering in path π,
- (ρ_Γ, σ, e) is consistent with ρ, and
- $e' = T[\![e]\!]\Gamma$ is the transformed version of e.

Then • $\rho_\Gamma, \sigma \vdash e' \Rightarrow v, \sigma''$ in pgm_Γ for some σ''.

Proof: By induction on the structure of the evaluation tree for $\rho \vdash e \Rightarrow v$ (which is the same as the structure of that for $\rho \vdash e \Rightarrow v, \pi$). The induction hypothesis in the proof for $\rho \vdash e \Rightarrow v$ is that the proposition holds for every judgement whose evaluation tree is a proper subtree of the evaluation tree for $\rho \vdash e \Rightarrow v$. Under the four assumptions stated above we will prove the conclusion: the transformed e' produces the same result v as e does.

Case $e \equiv x$ matches rule HE1. Clearly $\rho(x) = v$. On the one hand, if $x \notin \bigcup\Gamma$, then $e' \equiv x$ and $\rho_\Gamma, \sigma \vdash e' \Rightarrow v, \sigma$ as wanted because $\rho_\Gamma(x) = \rho(x)$ by the consistency assumption. On the other hand, if $x \in \gamma \in \Gamma$, then $e' \equiv x_\gamma$ and $\rho_\Gamma, \sigma \vdash e' \Rightarrow v, \sigma$ because $\rho(x) = \sigma(x_\gamma)$ by the consistency assumption.

Case $e \equiv f^i$ matches rule HE2. Clearly, $\rho \vdash e \Rightarrow f^i()$ and $e' \equiv e$, and so $\rho_\Gamma, \sigma \vdash e' \Rightarrow f^i(), \sigma$ as wanted.

Case $e \equiv A(e_1,...,e_a)$ matches rule HE3 with $a = \text{arity}(A)$. We have $\rho \vdash e_j \Rightarrow v_j, \pi_j$ for $j=1,..,a$ and $e' \equiv A(e'_1,...,e'_a)$ with $\rho_\Gamma, \sigma_{j-1} \vdash e'_j \Rightarrow v'_j, \sigma_j$ for $j=1,..,a$ where $\sigma_0 = \sigma$. Furthermore, $\pi = \pi_1...\pi_a$ and $v = A(v_1,...,v_a)$.

First we see that $(\rho_\Gamma, \sigma_{j-1}, e_j)$ is consistent with ρ for every $j=1,..,a$. For otherwise there is an $x \in \text{dom}(\rho)$ with $x \in \gamma \in \Gamma$ and a j with $1 \le j \le a$ such that $\rho(x) \ne \sigma_{j-1}(X_\gamma)$ and $x \in U_0(\pi_j)$. But since $\rho(x) = \sigma_0(X_\gamma)$ by the consistency assumption, there must be an $i < j$ such that evaluation of e'_j changes X_γ. Thus there is a variable $y \in \gamma$ such that $y \in DG(\pi_i)\Gamma$, so y interferes with x in $\pi_1...\pi_a$ which contradicts the non-interference of Γ. Thus by the induction hypothesis, $v'_j = v_j$ for all $j=1,..,a$, and so $\rho_\Gamma, \sigma \vdash A(e_1,...,e_a) \Rightarrow v, \sigma_a$ as desired.

Case $e \equiv e_1 \rightarrow e_2, e_3$ matches rule HE4 or HE5. It holds that $e' \equiv e'_1 \rightarrow e'_2, e'_3$. Assume $\rho \vdash e_1 \Rightarrow$ true which matches rule HE4; the other case is similar. We have $\rho \vdash e_1 \Rightarrow \text{true}, \pi_1$ and $\rho \vdash e_2 \Rightarrow v, \pi_2$ by the path semantics, and $\pi = \pi_1\pi_2$. By the induction hypothesis, $\rho_\Gamma, \sigma \vdash e'_1 \Rightarrow \text{true}, \sigma_1$ (for some σ_1) for the consistency with ρ of $(\rho_\Gamma, \sigma, e_1)$ follows from that of (ρ_Γ, σ, e). Furthermore, $(\rho_\Gamma, \sigma_1, e_2)$ must be consistent, or else evaluation of e'_1 has modified the value X_γ for some $x \in \gamma$ with $x \in U_0(\pi_2)$. But then evaluation of e_1 has defined some $y \in \gamma$. That is, $y \in DG(\pi_1)\Gamma$ and so y interferes with x in $\pi_1\pi_2$ contrary to the assumption that Γ is not interfering in π. We conclude that $(\rho_\Gamma, \sigma_1, e_2)$ is consistent with ρ, and by the induction hypothesis it follows that $\rho_\Gamma, \sigma \vdash e'_2 \Rightarrow v, \sigma_2$ (for some σ_2) with the correct value v as desired.

Case $e \equiv e_0 @ e_1$ matches rule HE6 or HE7. We must have $e' \equiv e'_0 @ e'_1$. We treat the cases of partial and full applications separately.

First the case of partial application. Assume $\rho \vdash e_0 \Rightarrow f^i(v_1,...,v_m), \pi_0$ with $m+1 < \text{arity}(f^i)$ so that rule HE6 applies. Then $\pi = \pi_0\pi_1$ where $\rho \vdash e_1 \Rightarrow v_{m+1}, \pi_1$ and $v = f^i(v_1,...,v_m,v_{m+1})$. We must show that v' equals v in the evaluation $\rho_\Gamma, \sigma \vdash e'_0 @ e'_1 \Rightarrow v', \sigma''$ of the transformed program. But $\rho_\Gamma, \sigma \vdash e'_0 \Rightarrow f^i(v_1,...,v_m), \sigma_0$ for some σ_0 by the induction hypothesis, for the consistency of $(\rho_\Gamma, \sigma, e_0)$ with ρ follows from the consistency of (ρ_Γ, σ, e). Then rule HIE7 applies:

$$\frac{\begin{array}{l}\rho_\Gamma, \sigma \vdash e'_0 \Rightarrow f^i(v_1,...,v_m), \sigma_0 \\ \rho_\Gamma, \sigma_0 \vdash e'_1 \Rightarrow v'_{m+1}, \sigma''\end{array}}{\rho_\Gamma, \sigma \vdash e'_0 @ e'_1 \Rightarrow f^i(v_1,...,v_m,v'_{m+1}), \sigma''} \qquad \text{where } m+1 < \text{arity}(f^i)$$

and we now just need to prove that $v'_{m+1} = v_{m+1}$ to get the desired conclusion that $f^i(v_1,...,v_m,v'_{m+1}) = v$. By the induction hypothesis, it suffices to show that $(\rho_\Gamma, \sigma_0, e_1)$ is consistent with ρ. But if $(\rho_\Gamma, \sigma_0, e_1)$ is not consistent with ρ, then there is an x used in evaluation of e_1 such that $x \in \gamma \in \Gamma$ and $\sigma'_0(X_\gamma) \ne \rho(x)$, and thus its value must have been changed during evaluation of e'_0. So evaluation of e_0 must define some $y \in \gamma$. Hence $y \in DG(\pi_0)\Gamma$ and $x \in U_0(\pi_1)$ and so y interferes with x in $\pi = \pi_0\pi_1$, which contradicts the assumption that Γ is not interfering in π.

Now to the full application case. Assume $\rho \vdash e_0 \Rightarrow f^i(v_1,...,v_m), \pi_0$ with $m+1 = a = \text{arity}(f^i)$ so that rule HE7 applies. Then $\pi = \pi_0 {<}{\downarrow}x^i_1...{\downarrow}x^i_m\pi_1\delta_1 \lozenge \pi'{>}$ where $\rho \vdash e_1 \Rightarrow v_{m+1}, \pi_1$ and $\rho' \vdash e^i \Rightarrow v, \pi'$ and $\delta_1 = \Delta(x^i_{m+1}, e_1)$. We must show that v' equals v in the evaluation $\rho_\Gamma, \sigma \vdash e'_0@e'_1 \Rightarrow v', \sigma''$ of the transformed program. By an argument similar to that used above, it holds that $\rho_\Gamma, \sigma \vdash e'_0 \Rightarrow f^i(v_1,...,v_m), \sigma_0$ (for some σ_0) and then rule HIE8 applies. The variable list of the called function f^i is $x^i_1,...,x^i_a$ and that of the transformed f^i in pgm_Γ is $\xi^i_1,...,\xi^i_a$.

$$\frac{\begin{array}{l}\rho_\Gamma, \sigma \vdash e'_0 \Rightarrow f^i(v_1,...,v_m), \sigma_0\\ \rho_\Gamma, \sigma''_0 \vdash e'_1 \Rightarrow v_{m+1}, \sigma'_1\\ \rho'_\Gamma, \sigma''_1 \vdash e'^i \Rightarrow v, \sigma''\end{array}}{\rho_\Gamma, \sigma \vdash e'_0@e'_1 \Rightarrow v, \sigma''} \qquad \begin{array}{l}\text{where } m+1 = a = \text{arity}(f^i),\\ e'^i \text{ is the body of } f^i \text{ in } \text{pgm}_\Gamma,\\ \sigma''_0 = \sigma_0[\xi^i_j \mapsto v_j \text{ for } j=1,..,a],\\ \sigma''_1 = \sigma'_1[\xi^i_{m+1} \mapsto v_{m+1}], \text{ and}\\ \rho'_\Gamma = [\,\xi^i_j \mapsto v_j \text{ for } j=1,..,a].\end{array}$$

Now letting $\sigma''_0 = \sigma_0[\xi^i_j \mapsto v_j \text{ for } j=1,..,a]$, the tuple $(\rho_\Gamma, \sigma''_0, e_1)$ is consistent with ρ. For else there is an $x \in U_0(\pi_1)$ with $x \in \gamma \in \Gamma$ which has been redefined either by a definition of some $y \in \gamma$ during evaluation of e_0, or by a push of some $x^i_j \in \gamma$ from the closure. In the first case $y \in DG(\pi_0)\Gamma$ and so y interferes with x, and in the second case $x^i_j \in DG({\downarrow}x^i_j)\Gamma$ and so x^i_j interferes with x in ${<}{\downarrow}x^i_1...{\downarrow}x^i_m\pi_1\delta_1 \lozenge \pi'{>}$ and hence in π. In either case the non-interference assumption is contradicted, and so $(\rho_\Gamma, \sigma''_0, e_1)$ must be consistent with ρ. From the induction hypothesis we conclude that v'_{m+1} equals v_{m+1} in $\rho_\Gamma, \sigma''_0 \vdash e'_1 \Rightarrow v'_{m+1}, \sigma'_1$.

It now only remains to be shown that with $\rho'_\Gamma = [\,\xi^i_j \mapsto v_j \text{ for } j=1,..,a]$ and $\sigma''_1 = \sigma'_1[\xi^i_{m+1} \mapsto v_{m+1}]$ we have $(\rho'_\Gamma, \sigma''_1, e'^i)$ consistent with $\rho' = [x^i_j \mapsto v_j \text{ for } j=1,..,a]$.

Condition 1 for consistency: for $x^i_j \in \text{dom}(\rho')$ with $x^i_j \notin \cup\Gamma$, we have $\xi^i_j = x^i_j \in \text{Var}$ and hence $\rho'_\Gamma(x^i_j) = v_j = \rho'(x^i_j)$.

Condition 2: assume x^i_j is used in evaluation of e'^i. For $x^i_j \in \text{dom}(\rho')$ and $x^i_j \in \gamma \in \Gamma$, we have $\xi^i_j = X_\gamma$. Now if $j = m+1$ then clearly $\sigma''_1(X_\gamma) = v_{m+1} = \rho'(x^i_{m+1})$. If on the other hand $j < m+1$ and contrary to our expectation, $\sigma''_1(X_\gamma) \neq \rho'(x^i_j)$, it must be because of an assignment to X_γ by e'_1 or because another parameter x^i_k also in γ has been defined after x^i_j (for some $k > j$ where $1 \leq k \leq m+1$). In the first case, $y \in DG(\pi_1)\Gamma$ for some $y \in \gamma$, and in the second case $x^i_k \in DG({\downarrow}x^i_k)\Gamma$. In either case $x^i_j \in U_0(\pi')$ and thus y (respectively x^i_k) interferes with x^i_j in ${<}{\downarrow}x^i_1...{\downarrow}x^i_m\pi_1\delta_1 \lozenge \pi'{>}$ which contradicts the assumption that Γ is not interfering in π.

This proves that $(\rho'_\Gamma, \sigma''_1, e'^i)$ is consistent with ρ', and so by the induction hypothesis that $\rho'_\Gamma, \sigma''_1 \vdash e'^i \Rightarrow v, \sigma''$ for some σ'': the correct result v is returned by the transformed expression e' in this case also.

This concludes the proof by induction on the structure of the evaluation tree for $\rho \vdash e \Rightarrow v$ and hence the lemma. □

This completes the proof of the globalization transformation. Now we will show that all the components in the method serve their purpose. Let an applicative H program pgm be given.

Proposition 5.2-2: The grammar G_{pgm} constructed for pgm by Algorithm 4.4-1 satisfies

$$\Pi(pgm) \subseteq L(G_{pgm}).$$

Proof: Similar to the proof of the corresponding Proposition 3.1.2-1. The proof of the case $e \equiv e_0 @ e_1$ requires the correctness result for the closure analysis (Proposition 5.1-1); a rule must be generated for each possible abstract closure value of e_0. □

The interference analysis ia presented in Section 3.2 can be applied to the grammars constructed by Algorithm 4.4-1, for the rule right hand sides have the form required for members of Rhs (defined in Section 3.1.1). The correctness of using the interference analysis on G_{pgm} is proved in

Proposition 5.2-3: For du-grammar G_{pgm} and variable grouping Γ for pgm, it holds that

$$ia(G_{pgm})\Gamma \supseteq Interf(pgm, \Gamma).$$

Proof: We have $ia(G_{pgm})\Gamma = IA[\![N_{pgm}]\!]\Gamma = ienv_\Gamma(N_{pgm}) \supseteq \cup \{ Interf(\pi, \Gamma) \mid N_{pgm} \to^* \pi \}$ by Lemma 3.2.3-1, and by Proposition 5.2-2, we have $L(N_{pgm}) = L(G_{pgm}) \supseteq \Pi(pgm)$ which proves the postulate. □

Now using Algorithm 3.3-1 to find an non-interfering variable grouping for pgm on the basis of ia is correct, for the correctness of the algorithm depends only on the correctness of the interference analysis which we have just proved.

Proposition 5.2-4: The variable grouping Γ constructed by Algorithm 3.3-1 is not interfering in pgm.

Proof: As for Proposition 3.3-1 but with reference to Proposition 5.2-3 instead of Proposition 3.2.3-1. □

Now the correctness of the entire globalization method for H can be stated.

Proposition 5.2-5: (Main correctness proposition for H). Let pgm be an applicative H program; let Γ_{non} be the variable grouping constructed by Algorithm 3.3-1; and let $pgm_{\Gamma_{non}}$ be the program constructed by Algorithm 4.5-1 when applied to pgm and Γ_{non}. Then $pgm_{\Gamma_{non}}$ is at least as strong as pgm.

Proof: Γ_{non} is not interfering in pgm according to Proposition 5.2-4, and therefore by Proposition 5.2-1, $\rho_0 \vdash pgm \Rightarrow v$ implies $\rho_0 \vdash pgm_{\Gamma_{non}} \Rightarrow v$. □

This concludes the demonstration of correctness of the globalization method for H programs.

6. RELATED WORK

This section compares our goals and approach to those of related work. Work by D. A. Schmidt on detecting global variables in denotational definitions inspired this study and deserves special mention. Schmidt's goal is to achieve better (more efficient) implementations from language definitions expressed in denotational semantics using a strict lambda calculus. He employs an analysis, based on the types of expressions, to detect whether all the store typed parameters in a language definition could indeed be replaced by a single global store structure.

U. Kastens and M. Schmidt derive an approach to lifetime analysis for procedure parameters from the attribute optimization techniques used in the compiler generator system called Gag. Their approach is very similar to ours and their goals are essentially the same. Their globalization method appears to be slightly weaker that ours and is not explicitly related to the (operational) semantics of the language under consideration.

A. Pettorossi discusses globalization in a strict first order language quite similar to our L. His technique adds "destructive markings" (annotations) to the program. The destructive markings govern explicit deallocation of store cells. If the store cell for a function parameter value is deallocated, the cell may be reused for a new value of the same (or another) function parameter.

Subsections 6.1 through 6.3 below detail these three approaches to globalization. Each is characterized by its application area, language, goal, main concepts, globalization criteria, transformation, and correctness considerations. In addition we try to assess each approach and compare it to our own.

Section 6.4 summarizes the characteristics of the three globalization methods mentioned and our own. Section 6.5 mentions related techniques used in compiler writing.

6.1 D. A. Schmidt: Detecting Global Variables in Denotational Specifications

This section is based on the paper [Schmidt 1985] and on Section 10.5.2 of [Schmidt 1986b].

6.1.1 Characteristics

The *application area* is that of semantics-directed compiler generation. This area is concerned with automatic construction of language implementations from formal language definitions. Usually, the definitions are denotational semantics specifications, expressed using a variant of the lambda calculus.

The *language* considered by Schmidt is a strict (call by value) version of the typed lambda calculus, and therefore is higher order. It has unspecified evaluation order unlike our H.

The *goal* of Schmidt's work is to alleviate some of the efficiency problems resulting from semantics-directed compiler generation. In particular, the objective is to obtain better object programs from automatically generated compilers. To this end, he develops criteria for detecting whether all store typed parameters in a denotational definition could be replaced by a single global variable. The globalizability of the store component is very important for run-time and storage efficient implementations of sequential languages such as Pascal: it is awkward to have several copies of the semantic store component at run-time. The criteria developed are useful also in designing optimizing compilers for applicative languages.

The *main concept* is that of single-threading: "A semantic definition whose store argument can be replaced by access rights to a single global variable while preserving operational properties is said to be *single-threaded* (in its store)." [Schmidt 1985, p. 300]. The paper gives sufficient criteria for single-threading, based on the types of the definition's expressions. To present the criteria we need two auxiliary definitions. A subexpression is *active* if it is not properly contained within an abstraction $\lambda x.M$. A store-typed expression is *trivial* if it is just a store-typed identifier occurrence. The sufficient single-threading criteria are [Schmidt 1985, pp. 304-305]:

"Definition. A closed expression F is single-threaded (in its store argument) if all subexpressions E of F possess the properties:

A. Noninterference
   (i) If E is store-typed, then if E contains multiple, disjoint active occurrences of store-typed expressions, then they are the same trivial identifier.
   (ii) If E is not store-typed, all occurrences of active store-typed expressions in E are the same trivial identifier.

B. Immediate evaluation
   (i) If $E = (\lambda x.M)$: $Store \to D$, then all active store-typed identifiers in M are occurrences of x.
   (ii) If $E = (\lambda x.M)$: $C \to D$, and C is not store-typed, then M contains no active store-typed expressions."

The single-threading criteria have two parts, noninterference and immediate evaluation. Noninterference corresponds to our concept with the same name, and immediate evaluation corresponds to non-enclosure. The use of the criteria can be shown by a few examples [Schmidt 1985, p. 305]:

"(a) $C[C_1; C_2] = \lambda s.\ C[C_2](C[C_1]s)$
(b) $E[E_1 + E_2] = \lambda s.\ E[E_1]s\ \mathit{plus}\ E[E_2]s$
(c) $P[\textbf{procedure C using I}] = \lambda n.\ \lambda s.\ C[C](\mathit{update}[I]n\ s)$
   where P: Procedure $\to$ *Nat* $\to$ *Store* $\to$ *Store*
(d) $M[\textbf{plus1}] = \lambda n.\ \mathit{succ}\ n$
   where M: Operator $\to$ *Nat* $\to$ *Nat*
(e) $C[C_1\ \textbf{op}\ C_2] = \lambda s.\ C[C_1]s\ \mathit{combine}\ C[C_2]s$
(f) $E[\textbf{valof C is E}] = \lambda s.\ E[E](C[C]s)$
(g) $Q[\textbf{procedure C}] = \lambda s.\ \lambda s'.\ C[C]s$
   where Q: Procedure $\to$ *Store* $\to$ (*Store* $\to$ *Store*)
(h) $C[\textbf{call P(E)}] = \lambda s.\ F[P](\lambda n.\ E[E]s)s$
   where F: Procedure $\to$ (*Nat* $\to$ *Nat*) $\to$ *Store* $\to$ *Store*

Expressions (a)–(d) are single-threaded. In particular, expression (a) complies with property (Ai), for its store-typed expressions are nested, not disjoint. The call-by-value reduction strategy forces a lock-stepped evaluation. Expression (b) shows the proper use of multiple access rights, complying with property (Aii), and expression (c) complies with property (Bi), for the occurrence of the store variable s is properly bound to the enclosing λs, guaranteeing that the call-time store will be used with the procedure upon invocation. Expression (d) clearly satisfies property (Bii). Property (Ai) is violated by expression (e), for the disjoint active occurrences of $C[C_1]s$ and $C[C_2]s$ suggest that C_1 and C_2 will each need local stores to properly complete the noninterfering evaluation. Expression (f) violates property (Aii), which creates a problem when it is used in an expression such as [(**valof** (C) **is** E_1) + E_2]: a local store for E_1 is needed. Expression (g) violates property (Bi), for the procedure object saves its declaration-time store for evaluation and ignores its call-time one. Finally, expression (h) violates property (Bii), for the 'thunk' ($\lambda n.$ E[E]s) created at procedure call time saves the call-time store for later use."

If we translate the expressions (a) through (h) into H, we can test our criteria on them. (The above examples are not dependent on the evaluation order being unspecified). We find that the *s* variables are not interfering in any of (a) through (d), assuming that evaluation of **E** cannot define *s*. Furthermore, *s* is not enclosed during evaluation of these expressions. But *s* is interfering in (e) under the (reasonable) assumption that evaluation of **C** may define *s*. In example (f) we observe that if **C** may define *s*, then so may **E**, and this is inconsistent with the assumption just made about **E**. Expressions (g) and (f) both enclose a store variable.

The *transformation* of a single-threaded denotational definition works as follows. The definition is assumed to use a *store algebra* that contains all functions working on the store type. All manipulation of store values is ultimately done by these store algebra functions. Now the definition is transformed by replacing the store algebra by a *store class* in the Simula sense. The store class contains one store variable and a collection of operations that access or modify this variable. Each function from the store algebra that has a store-typed parameter is replaced by an operation that accesses the store variable; and each function from the algebra that returns a store-typed value is replaced by an operation that modifies the store variable. The transformed definition uses the store class operations instead of the store algebra functions but is not otherwise modified. The net effect is that evaluation of a non-trivial store-typed expression assigns the result (the new store value) to the global store variable, and all references to store-typed values are made to the global store variable.

The *correctness* of the transformation is proved by an interpreter equivalence proof which is outlined but not given in the paper [Schmidt 1985]. It is not easy to see intuitively why the single-threading criteria are reasonable.

6.1.2 Assessment: D. A. Schmidt

In Schmidt's framework the type consistency requirements restrict the way expressions can be put together and the way they can behave. That is why the apparently very "local" single-threading conditions are sufficient. Our du-paths must explicitly tell what happens in subexpressions and in functions called from the expression under consideration. In the typed lambda calculus, this information is provided by the types of expressions and function calls. This approach would not work if coercion of the store type into another type or vice versa were allowed: the type system must be completely unforgiving.

Schmidt's criteria for single-threading are elegant and easily checkable. Why then have we developed another technique? There are several partial reasons for this:

1. Schmidt's language has no conditional expression and the inclusion of one requires numerous refinements to the single-threading concept. In the lambda calculus a non-strict conditional can be simulated using a strict basic function if(x,y,z) and letting y and z be function-valued expressions, but this is not possible in a first order language.
2. The use of types of expressions is highly "intensional". It is very important that expressions are given types in a certain way for a definition to be single-threaded in its store argument. In fact, types automatically assigned by a straightforward type system may fail to satisfy the single-threading criteria; extra conditions on the type system, or some human intervention, is needed to obtain single-threading.
3. Therefore it is not clear how the approach should be extended to apply to an untyped language.
4. The set of store-typed variables in a program plays the role of a single variable group in our framework. To find several variable groups would require detection of single-threading in several "store" types at the same time. While this may be feasible, it is not clear which requirements it would impose on the type system. Thus it is not clear how the approach could be extended to detect several globalizable variable groups.
5. The criteria do not rely on any particular evaluation order and thus may yield too conservative results in case a particular evaluation order could be guaranteed. It is not clear how a specific evaluation order could be taken into account.
6. Schmidt requires the store type to be non-functional. It is not clear to us whether this is essential for the validity of the single-threading criteria.

In conclusion, Schmidt's approach is simple and elegant for the typed lambda calculus, but it would require rather comprehensive modifications to apply it to an untyped functional language with a non-strict conditional, for example.

6.2 U. Kastens and M. Schmidt: Lifetime Analysis for Procedure Parameters

This section is based on [Kastens, Schmidt 1986]. Their approach originates from optimization of attribute evaluation in compilers generated by the compiler generator system Gag [Kastens, Hutt, Zimmermann 1982], [Farrow, Yellin 1986], [Kastens 1987]. The approach is quite similar to ours and came to our attention only rather late in the project, after we had independently explored the concepts of definition-use path, grammars, and globalization criteria to some extent.

6.2.1 Characteristics

The *application area* is optimizing transformations of procedure calls. The *language* is a kind of extended Pascal; procedures have input parameters and output parameters. Function results are returned via output parameters. The language is higher order; functional parameters are allowed. (But the techniques seem to be insufficient to handle the higher order case properly).

The *goal* of the work is to replace procedure parameters (input as well as output) by global variables whenever this is possible. The techniques can also be used to improve calling sequences, thus yielding optimization of tail recursive calls as a special case.

The concepts of *lifetime analysis* and *lifetime grammar* are central. The lifetime grammar is constructed from the program, and the lifetime analysis is done on the grammar as in our approach. To begin with, consider the restricted case of first order programs without nested procedures. The grammar for a program has a nonterminal symbol for each procedure in the program. The strings derivable from the grammar are comparable to our du-paths (Section 2.3). They contain terminal symbols D_x for definition of parameter x and U_x for use of x, corresponding to our $\downarrow$x and $\uparrow$x. There is no counterpart of our copy action y»x or of the call structure brackets "<", "◊", and ">". However, "If several calls of the same procedure occur in one context or if a procedure is called directly recursive then references to their objects [that is, variables or parameters] are distinguished by number indices, ..." [Kastens, Schmidt 1986, p. 56]. The purpose of this probably is the same as that of our brackets: to distinguish different invocations of the same function and hence different incarnations of the same variable.

To find the lifetime of an object, only its definition and its last use need be known. Each rule right hand side is processed by deleting all but the last use U_x for each x and replacing this last one by the symbol L_x. (This is meaningful because each right hand side represents precisely one procedure body, that is, a single lexical scope). Now the lifetime of one incarnation of object x begins with a D_x and ends with the first L_x following that. The resulting grammar G_L may be "projected" onto a set B of variables (which plays the role of a variable group in our terminology). Projection means that every symbol D_x with $x \in B$ is replaced by D, and the D_x for which $x \notin B$, are delected. The use symbols U_x are treated similarly. The projected grammar is called G_B.

The *globalization condition* is that definitions and uses alternate in every string derivable from the grammar: $L(G_B) \subseteq (DL)^*$. Intuitively, the last use of an object must precede the definition of its next incarnation.

The above technique works for first order and nonnested procedure definitions. A local definition (within another definition) can be made global by introducing a new parameter for each non-local variable. This way nested procedure definitions can be made nonnested, and so the technique applies also to (first order) nested definitions.

The extension to higher order programs (that is, with procedures passed via input or output parameters) is more problematical. According to the paper it is sufficient to add "chain productions" to the grammar [Kastens, Schmidt 1986, p. 58]:

"Let p be a procedure with a functional parameter f, and q the name of an arbitrary declared or formal procedure. If f is an input parameter a call of p may have the form $p(q)$ and f may be called (or further passed as a parameter) in the body of p. If f is an output parameter the body of p contains parameter assignments of the form $f := q$ and f may be called (or further passed as a parameter) after a call of p. For the construction of our lifetime grammar G_L calls of f are treated like calls of declared procedures: They are transformed into occurrences of the nonterminal f. For each parameter assignment to f (input as well as output) a chain production $f ::= q$ is added to G_L."

This approach however is too simple to work properly in general. Consider the example program below in which one functional parameter is applied to another (such cases will often result by flattening a nested higher order program):

```
proc p(proc pf, pg)
begin pf(pg) end;

proc q(proc qf)
begin qf end;

proc r
begin ... end;

p(q, r)
```

The "chain production" approach as described would yield chain productions $pf ::= q$ and $pg ::= r$, which is correct. But the rule $qf ::= pg$ must also be added, and this requires that we know that q is a possible value of pf. This information can be collected by a simple version of our closure analysis (Section 4.3), or by rewriting pf using the grammar rules already produced (as in the flow analysis described in [Jones 1987]).

Another problem with the extension to higher order programs is the adding of indices to terminal symbols in the grammar. In the expression $f(...) + g(...)$, f and g may be functional parameters bound to the same procedure p, and thus indices should be added to the grammar rule symbols for this expression. But discovering that f and g might denote the same procedure would again require a closure analysis or a rewriting of f and g using the grammar chain productions. We conclude that the lifetime grammar technique can probably be extended to higher order programs but that the extensions have not been presented in complete detail in the paper.

The *globalization transformation* simply replaces a set B of variables for which $L(G_B) \subseteq (DL)^*$ by a single global variable X. First, assume $x \in B$ is an input parameter of procedure p. In p's body, every occurrence of x is replaced by X, and in each call to p, the argument expression corresponding to x is removed and an assignment to X is inserted *before* the call. Second, assume x is an output parameter of p. Then the assignments to x in p's body are replaced by assignments to X, and in each call to p, the argument expression corresponding to x, which must be an occurrence of some variable y, is removed and an assignment y := X is inserted *after* the call.

The *correctness* of this approach to globalization is not discussed in the paper. The construction of lifetime grammars is not formalized and is not (explicitly) based on assumptions about the semantics of the language. However, the globalization condition is simple and intuitively reasonable, and it is clear that the approach could be formalized if desired.

6.2.2 Assessment: U. Kastens and M. Schmidt

The globalization criteria are much simpler and probably easier (and more efficient) to apply than ours. Also, the lifetime grammar constructed for a program is likely to be more compact than our du-grammars. However, the technique appears to be somewhat more conservative than ours. One reason is the lack of copy actions. Consider the exp function in our μP interpreter (Example 2.1-2):

```
exp(e,se) =
     e= constant "k"  → k,
     e= variable "z"  → lookup(z,se),
     e= "e1+e2"       → plus(exp(e1,se),exp(e2,se)),
     ...
```

The lifetime grammar for this could be (compare with the du-grammar in Example 3.1.2-2):

$$\begin{aligned} exp ::= {} & U_e\,U_e \\ \mid {} & U_e\,U_e\,U_e\,U_{se} \\ \mid {} & U_e\,U_e\,U_e\,U_e\,D_e\,U_{se}\,D_{se}\ exp\ U_e\,D_e\,U_{se}\,D_{se}\ exp \end{aligned}$$

Deleting all but the last U in each rule and replacing the last one by L, we obtain

$$exp ::= L_e \mid L_e\,L_{se} \mid D_e\,D_{se}\ exp\ L_e\,D_e\,L_{se}\,D_{se}\ exp$$

Finally, projecting this grammar onto B = {se}, we get

$$exp ::= \varepsilon \mid L \mid D\ exp\ L\ D\ exp$$

Repeatedly choosing the last alternative in this rule will derive a substring of form ...DDD..., and so the set of strings generated by the grammar cannot be a subset of (DL)*. Therefore Kastens and Schmidt's globalization condition is not satisfied for variable se of function exp, but our analysis in Example 3.3-1 found that se *is* indeed globalizable (together with variable sc). The problem is that the definitions D_{se} of se are not definitions at all; they are copies se»se in our terminology.

It does not seem to be easy to extend this approach to include copy symbols. For correctness of the globalization condition, it is essential that there is at least one D_x for every L_x in each string derivable from the grammar, and this would not be the case if definitions could be deleted when they are just copies. However, to do justice to this approach we should point out that copies which are not definitions are less likely to occur in imperative programs than in purely applicative ones.

The lack of call structure brackets is another reason for the approach being conservative. This can be helped by a small trick, however. Consider the contrived program

```
proc f(x, y)
begin  if y = 0 then a := x else f(3, y–1) end
```

Informally it is easy to see that x is globalizable, but it fails to satisfy the globalization condition. The lifetime grammar is

$$f \quad ::= \; L_y\, L_x \;\mid\; D_x\, L_y\, D_y\, f$$

and projected onto B = {x}, it is

$$f \quad ::= \; L \;\mid\; D f$$

which can derive the substring ...DDD... The (minor) problem here is that x is not used in the second branch of the conditional. The analysis will work properly if an artificial last use L_x of x is added at the beginning of the second grammar rule alternative. This observation is made also in [Kastens 1987] in the context of lifetime analysis for attribute grammars.

In conclusion, this approach is simpler and easier to understand than ours, but at the cost of being somewhat more conservative. Furthermore, the approach is not readily applicable to higher order languages.

6.3 A. Pettorossi: Recursive Programs Which Are Memory Efficient

This description is based on [Pettorossi 1978] and [Pettorossi 1984].

6.3.1 Characteristics

The *application area* is transformation of recursive programs. The *language* is first order recursion equations with strict application and left to right evaluation, very similar to our L. The implementation of the language must work with a pool of store cells (locations), each marked "active" or "inactive" according as the cell is in use or not. All intermediate values in an evaluation must reside in a store cell. A parameter is passed by reference if the argument expression is a variable, by value otherwise. Notice that this is not the stack-based implementation of first order recursive programs we have assumed in the rest of the report.

The *goal* of the work is to reduce the storage requirements of recursive programs. This is done by explicitly marking store cells "inactive" as early as possible, thus freeing them to hold other values.

A *central concept* is that of destructive marking. A destructive marking is an annotation of an operator that tells which of its argument values may be discarded after evaluation of the operator. A marking is a tuple <010...0> of ones and zeroes describing the destructibility of each argument position of the operator. A "1" means that the value in the corresponding argument position may be discarded after evaluation of the operator, and a "0" means that it cannot be discarded (yet). Discarding a value is done by marking its store cell "inactive". For example, $\text{times}_{<10>}(x, y)$ means that the value of x may be discarded while that of y must be retained. Thus the result of the multiplication may be stored in the cell formerly occupied by x. Operators are basic functions (plus, times, ...) or the conditional (if).

The destructive markings must be correct for the program; the value of a variable must not be destroyed before its last use.

A program is *transformed* by adding destructive markings to every operator occurrence. This is done as follows. To begin with, each operator is marked with <0...0>, that is, no arguments at all may be destroyed. This is clearly safe. Then 0's are changed into 1's, one at a time, as long as this can be done while preserving the meaning of the program. This is done using two functions called *correct* and *inspect*. The former is used to check whether an attempted change of a single 0 into 1 would preserve the meaning of the program. The latter computes an approximation (probably) to the set of variables that evaluation of a given term may destroy.

As to *correctness*, the semantics of programs with destructive markings is defined formally, and correctness requirements for destructive markings are given in [Pettorossi 1978]. Also some requirements on the functions *correct* and *inspect* are given. However, these important functions are not defined in the papers, either formally or informally, and thus the correctness of the marking procedure is not proved.

6.3.2 Assessment: A. Pettorossi

Pettorossi's approach is different from the others we have considered. The other approaches would replace a function parameter by a fixed global variable in a fixed location; it is statically decided which location the variable will occupy (namely, that of the global variable). In this framework, this is not the case. Each new incarnation of a variable may be put into a different store cell, for example one just freed by destroying another value.

This yields the effect of "dynamic variable groups": in one computation, variable x may use the same cell as y but not the same as z, and in another computation, x and z may use the same cell, never used by y. This allows for more optimal storage reuse than the other, more static, globalization concepts. On the other hand, Pettorossi's approach presupposes a heap-like store, which is rather expensive in terms of run-time for a first order language (due to management of the "active" tags on store cells).

Pettorossi's method also requires that composite values (those that require more than one store cell) do not share. That is, destruction of one structured value must not affect any other value. This restriction is lifted in Mycroft's Ph. D. thesis [Mycroft 1981].

6.4 Overview

Below we give in compact tabular form an overview of the three globalization methods in addition to our own.

Topic	Schmidt (Section 6.1)	Kastens & Schmidt (Section 6.2)	Pettorossi (Section 6.3)	Sestoft
Language	Lambda-calculus, strict, higher order, unspecified eval. order	"Extended Pascal procedures", strict, higher order?, left-to-right	First order recursion equations, strict, left-to-right	Strict first/higher order recursion equations, left-to-right
Globalization criteria	Single-threading, based on the types of expressions	Disjoint lifetimes, analysis of lifetime grammar	"Destructive markings" are provided by an analysis, not given in detail	Interference in du-paths, interference analysis of du-grammar generated from path semantics
Variable groups	The store type corresponds to a single variable group and must be chosen manually	Several variable groups possible, automatically detected	Locations are explicitly deallocated as soon as possible, "dynamic variable groups"	Several variable groups possible, automatically constructed
Transformation	Store algebra is replaced by a store class that has a single global var. of type store	Stack pushes are replaced by assignments	"Destructive markings" governs explicit deallocation of store cells	Stack pushes are replaced by assignments
Correctness	Interpreter equivalence proof: "tracking proof"	Not formally stated	Correctness requirements are stated but not proved	Proof on (operational semantics) evaluation trees
Advantages	Easy check on typed expressions, nice transformation	Quite simple and efficient analysis, works for untyped languages, adaptable to eval. order	Works for untyped languages, deallocates early	Discovers many opportunities for globalization, works for untyped languages, adaptable to evaluation order
Drawbacks	Hard to automate type assignment or extend to untyped lang., not adaptable to eval. order	Conservative in some cases, not quite adequate for higher order languages	Only first order languages, expensive store model	Costly analysis (?), complicated interference criteria

6.5 Other Work

A notion of *path semantics* for a first order language with call by need is found in [Bloss, Hudak 1988]. A path through a function expresses the order of evaluation of the (delayed) argument expressions of the function. Thus the concept of a path is related to but not identical to ours. Their path semantics is expressed in a denotational style, and an approximation is used for strictness analysis and other optimizations.

Our concept of *interference* is somewhat related to that of [Reynolds 1978]. His concept applies to imperative languages: two program phrases interfere if they may access and modify the same structures. Thus the effect of parallel execution of interfering program phrases is unpredictable in general.

In Section 1.4 we discussed other related work (compiler optimizations).

7. EVALUATION AND RESULTS

We can measure the quality of our globalization method (developed in Sections 2 through 5) along the following three dimensions:

1. How good are the globalization criteria?
2. How expensive are the analyses and transformations to apply?
3. How much efficiency can be gained by using the globalization method?

7.1 Quality of the Globalization Criteria

The globalization criteria have two parts for higher order languages: interference and enclosure; only the first applies to first order languages. Here we focus on the interference criteria as developed in Section 2.4. The problem of enclosure in the present simple H is trivial because H has only single applications e_0 @ e_1. The problem of a reasonable enclosure analysis for a version of H with multi-applications is addressed in Section 8.1.2.

The interference criteria perform quite well on the examples that we have applied them to. We have not found that our criteria fail in any cases where globalizability is intuitively "obvious". Also, we have not found any cases where our globalization criteria are more conservative than those of [Schmidt 1985]. The comparison in Section 6.2 shows that our criteria are (strictly) less conservative than those of [Kastens, Schmidt 1986].

It would be interesting to make formal comparisons between Schmidt's criteria and ours. This would require that we formalize the translation of a denotational definition def in the lambda calculus into an H program pgm (by lambda-lifting [Johnsson 1985]). The types of expressions in def must carry over to pgm. Then if def is single-threaded in its store type, we let γ be the variable group containing all store-typed variables in pgm, and let $\Gamma = \{ \gamma \}$ be the variable grouping containing only γ. Then we must show that Γ is not interfering in pgm according to *our* interference concept. We shall not attempt to do this here, however.

7.2 Expected Cost of the Analyses and Transformations

The cost of doing the analyses and transformation is hard to judge. The algorithms given for the du-grammar construction and the globalization transformation can be implemented quite efficiently without much modification. On the other hand, the interference analysis ia (Algorithm 3.2.2-1) should probably be improved, along with the related analyses ua_0 and da_Γ (Section 3.2.1). The same holds for the closure analysis (Section 4.3). Simple straightforward implementations of these analyses will have a worst-case run-time which is exponential in the size of the programs. More sophisticated implementations may be feasible, however.

Furthermore, the average behaviour on "typical" programs may be much better than the worst-case behaviour. In practice, the theoretical worst-case behaviour may not be very significant. An experimental implementation of our techniques would give valuable information on this point, and would allow us to focus improvement efforts on the most critical parts of the system (that is, those that are slowest in practical usage).

7.3 Benefits to Be Expected from Globalization

The run-time and storage savings of globalization can be expected to be quite substantial in a language like Pascal because values do not share and so stacking a value (for a call by value parameter) requires a new complete copy of it. Run-time improvements of 25 per cent are reported in [Kastens, Schmidt 1986].

On the other hand, in languages such as Lisp or Scheme where structured values may share, passing a value parameter to a function requires only that a pointer is passed. Therefore one cannot expect that globalization *in itself* will give any substantial run-time or storage benefits.

We have made a small experiment (using the Franz Lisp system and the Chez Scheme system, running compiled code on a VAX11/785) which confirms this. An applicative first order interpreter int for a tiny language similar to μP from Example 2.1-2 was implemented (in Lisp and in Scheme). It contained a number of store variables (corresponding to `sc` and `se` in the example). Then we transformed int into an imperative program int_Γ in which the store variables were replaced by a single global store variable (corresponding to `S` in the example). The resulting program is similar to that given at the end of Example 2.1-2. Running int_Γ (instead of int) will avoid passing the current store value as parameter, and thus should save some run-time and storage. However, as said above, only a pointer is passed which is a relatively inexpensive operation. And in fact the two versions, int and int_Γ, have identical run-time and storage consumption for a range of inputs: globalization gives no improvement in this case.

Globalization makes another optimization possible, however. Once all the store variables in the interpreter are replaced by one global variable `S`, we can replace those "basic functions" (`update`, `lookup`, ...) that work on the store by *destructive* versions. Hence we may replace the basic function `update` by a procedure that updates the value of `S` destructively (using `rplaca` in Lisp or `set-car!` in Scheme, for example). This modification substantially reduces the storage requirements and the need for garbage collection in our experiments: by around 70 per cent. Note that this corresponds to the replacement of a store algebra by a store class as in [Schmidt 1985].

For bigger and more realistic programs we expect that globalization in itself will yield reasonable run-time and storage improvements in contrast to these toy examples. Furthermore, the effect of the "destructive" update optimizations can be expected to be considerable. Notice that it is safe to use destructive functions (such as `set-car!`) here, even though it is usually considered a bad or unsafe programming style.

8. DIRECTIONS AND OPEN PROBLEMS

The present work may be improved and extended in various directions. Section 8.1 first presents three minor technical improvements. Then Section 8.2 discusses extension and alternative uses of path semantics. In particular, can our approach be used to detect when heap allocated objects may be stored on a stack?

All of this work concerns strict languages, but lazy languages currently receive much attention. Section 8.3 discusses the prospect for applying our approach to the analysis of lazy languages.

8.1 Improvements of This Work

A few small improvements of the present work will be suggested here.

8.1.1 Consistent Presentation

The imperative parts of the language L could employ the same syntax and mechanisms for assignment that H does. This would simplify the definition of L and make Sections 2.2.3 and 4.1.2 more parallel. At present, in L we write `f([X:=7]),9)` if `f` is defined by `f(y)=...`, and in H's concrete syntax we would write `f 7 9` where `f X y=....` The H notation without the explicit assignment operator is the more elegant and should be used for L also.

With this change, the globalization transformations for L and H could be made almost identical. This further allows us to make their proofs (in Sections 3.4 and 5.2) structurally similar. This improvement would require a change in the definition of imperative L, the L globalization transformation, and its proof. In particular, the proof would become shorter but would have more complicated induction steps, like the proof for H. This change has not been done for lack of time.

8.1.2 Multi-Applications in H

The higher order language H should be extended with multi-applications e_0 `@` $(e_1, ..., e_n)$ to avoid unnecessary enclosure of variables, as discussed at the end of Section 4.1.1.

A multi-application expression is evaluated by first evaluating e_0 to obtain a closure $f^i(v_1, ..., v_m)$. (Recall that $m < \text{arity}(f^i)$). The further action depends on whether the total number $m + n$ of arguments is less than, equal to, or greater than, the arity $a = \text{arity}(f^i)$ of f^i:

(1) If $m + n < a$, then e_1 through e_m are evaluated to obtain values $v_{m+1}, ..., v_{m+n}$, and a new extended closure $f^i(v_1, ..., v_m, v_{m+1}, ..., v_{m+n})$ is returned.

(2) If $m + n = a$, then e_1 through e_n are evaluated to obtain values $v_{m+1}, ..., v_{m+n}$, a new environment is built that binds $x^i_1, ..., x^i_a$ to $v_1, ..., v_m, v_{m+1}, ..., v_{m+n}$, respectively, and the body e^i of f^i is evaluated in this new environment.

(3) If $m + n > a$, then sufficiently many argument expressions are evaluated to provide f^i with all its arguments, a new environment is built, and the body of f^i is evaluated. The result is a new closure which is applied to the remaining argument expressions. That is, in this case the expression e_0 `@` $(e_1, \ldots, e_n)$ is evaluated just as $(e_0$ `@` $(e_1, \ldots, e_{a-m}))$ `@` $(e_{a-m+1}, \ldots, e_n)$.

This way of evaluating multi-applications is rather similar to the "DISPATCH k" instruction of the G machine for evaluation of lazy supercombinator programs [Peyton Jones 1987, pp. 371-376].

The imperative extension of H should be extended with multi-applications also. They would be evaluated as for applicative H with the natural modifications for assignment. In case (1) above, no assignments are done. In cases (2) and (3), assignments may take place as soon as each argument expression $e_1, \ldots$ is evaluated. Alternatively, all the assignments may be done at the time of building the new environment. The new H path semantics must reflect the choice.

Introduction of multi-applications requires modification of Sections 4 and 5, but the changes are straightforward. They concern the syntax and semantics for applicative and imperative H, path semantics, closure analysis, grammar construction, and the globalization transformation.

Assuming that the closure analysis ca (Section 4.3) is modified to work for multi-application H, we can use it to find those variables that *may* be enclosed during an evaluation. This is useful, for variables that are never enclosed are particularly interesting candidates for globalization; they cannot be allocated on the heap. Let pgm be an H program (with multi-applications). We intend that Enclosed(pgm) is a superset of the set of variables that may be enclosed in an evaluation of pgm.

Definition 8.1.2-1: The enclosure analysis Enclosed(pgm) for an H program pgm with multi-applications is defined as follows (where HMExpr is the set of expressions in H extended with multi-applications):

$$\text{Enclosed(pgm)} = \bigcup \{ EC[\![e^i]\!] \mid i \in I \}$$

where $EC: \text{HMExpr} \to \wp(\text{Var})$

$$\begin{aligned}
EC[\![x]\!] &= \{\} \\
EC[\![f^i]\!] &= \{\} \\
EC[\![A(e_1,..,e_a)]\!] &= EC[\![e_1]\!] \cup \ldots \cup EC[\![e_a]\!] \\
EC[\![e_1 \to e_2, e_3]\!] &= EC[\![e_1]\!] \cup EC[\![e_2]\!] \cup EC[\![e_3]\!] \\
EC[\![e_0 \,@\, (e_1,..,e_n)]\!] &= EC[\![e_0]\!] \cup EC[\![e_1]\!] \cup \ldots \cup EC[\![e_n]\!] \\
&\quad \cup \bigcup \{ \text{app}(c, n) \mid c \in ca(e_0) \}
\end{aligned}$$

where $\text{app}: \text{AClo} \times \{0, 1, 2, \ldots\} \to \wp(\text{Var})$

$$\begin{aligned}
\text{app}((f^i,m),n) &= \{ x^i_{m+1}, .., x^i_{m+n} \} && \text{if } m + n < a \\
&= \{\} && \text{if } m + n = a \\
&= \bigcup \{ \text{app}(c', m+n-a) \mid c' \in ca(e^i) \} && \text{if } m + n > a
\end{aligned}$$

where $a = \text{arity}(f^i)$ □

The auxiliary function app is well-defined because all arities are non-negative and because $m + n - a < n$ which follows from the property that $m < \text{arity}(f^i)$ for all closures (f^i,m).

With this analysis a variable x of pgm is an interesting candidate for globalization if $x \notin \text{Enclosed}(pgm)$ and $(x,x) \notin ia(G_{pgm})\Gamma$. An "enclosure analysis" like the one above could also be designed for the (single application) version of H discussed in Section 4, but its results would be rather trivial: all variables except the last one of each function will always be enclosed, and the last one will never be enclosed.

8.1.3 Tail Recursion Detection

As noted in Section 2.1, our techniques will not detect all tail recursive calls because this might require the introduction of temporary variables in general. A small change to the path semantics will allow the detection of all tail recursive calls (in the case of the first order language L). The path semantics rule for a function call $f^i(e_1,...,e_a)$ must be changed to reflect the assumption that temporary variables are introduced during evaluation of argument expressions whenever needed. This is done by moving all the definition (or copy) actions to the end of the prelude in a call structure: the path $<\pi_1\delta_1...\pi_a\delta_a \lozenge \pi>$ becomes $<\pi_1...\pi_a\delta_1...\delta_a \lozenge \pi>$. The new path semantics rule which replaces rule PE5 is

$$\frac{\rho \vdash e_j \Rightarrow v_j, \pi_j \quad \text{for } j=1,...,a \qquad \rho' \vdash e^i \Rightarrow v, \pi}{\rho \vdash f^i(e_1,...,e_a) \Rightarrow v, <\pi_1..\pi_a\delta_1..\delta_a \lozenge \pi>} \qquad \begin{array}{l} \text{where } a = \text{arity}(f^i), \\ e^i \text{ is the body of function } f^i, \\ \pi' = [x^i_1 \mapsto v_j,...,x^i_a \mapsto v_a], \\ \delta_j = \Delta(x^i_j, e_j) \text{ for } j=1,...,a. \end{array}$$

Naturally, the semantics for imperative L should be modified to implement the assumption about introduction of temporary variables. However, the new semantics rule for this cannot be expressed elegantly unless the notation for assignment in L is modified as noted in Section 8.1.1.

8.2 Other Uses of Path Semantics

In this section we propose some possible uses for suitably modified concepts of path semantics. The idea is to reuse the development of this report for other purposes, in the hope that the grammar constructed from a program and its path semantics may have other uses besides that of interference detection. We can use path actions other than $\uparrow x$, $\downarrow x$, and $y \gg x$, design an exact analysis applicable to such modified paths, and derive an abstraction of this new analysis which is applicable to the grammar. The ideas presented here are very preliminary, and some of the applications may turn out to be infeasible.

8.2.1 Sharing Analysis

Assume our (first order strict) language L is extended with operators `cons`, `car`, and `cdr` similar to those of Lisp or Scheme. That is, the result of evaluating `cons(x,y)` is a composite value (structure) which has the values of x and y as substructures. If now this value is bound to variable z, then z and x (as well as z and y) *share* a substructure. Hence modification of the structure bound to x may modify the structure bound to z, and vice versa. Changing the structure of a variable such as x by modifying the structure bound to it (instead of binding a new structure) is called destructive update [Mycroft 1981], [Pettorossi 1978, 1984].

Destructive updates are desirable because they conserve space and garbage collection. But clearly, a destructive update of variable x is admissible only if x does not share its value or any part of it with other variables. Therefore *sharing analysis* of a program is interesting; it is useful to know when a variable is guaranteed not to share with others.

It may be feasible to do a sharing analysis as in [Mycroft 1981] by using a modification of the path concept and the path semantics. To see how this *might* be done we define a new kind of path, called substructure paths (ss-paths for short). The path actions are uses: $\uparrow x$, substructure definitions: $\pi \gg x$, and call structures: $<\pi_1 \gg x^i_1 \ldots \pi_a \gg x^i_a \lozenge \pi>$. The $\pi \gg x$ action means that variable x gets a new value which shares (or may share) with the variables with uses in π. Thus

$$\begin{aligned} \text{Use} &= \{ \uparrow x \mid x \in \text{Var} \} \\ \text{SSPath} &= \text{Use} \cup \text{SSPath}^* \cup \{ <\pi_1 \gg x^i_1 \ldots \pi_a \gg x^i_a \lozenge \pi> \mid \pi_j \in \text{SSPath},\ x^i_j \in \text{Var} \}. \end{aligned}$$

A path semantics may be defined that gives the ss-path π for an expression e in addition to its value v. The ss-path semantics will have special rules for `cons`, `car`, and `cdr`. A substructure path analysis SU that finds the set of variables $SU[\![\pi]\!]$ that may share with v can be defined as follows:

$$\begin{aligned} &SU: \text{SSPath} \to \wp(\text{Var}) \\ &SU[\![\uparrow x]\!] &&= \{ x \} \\ &SU[\![\pi_1 \ldots \pi_a]\!] &&= SU[\![\pi_1]\!] \cup \ldots \cup SU[\![\pi_a]\!] \\ &SU[\![<\pi_1 \gg x^i_1 \ldots \pi_a \gg x^i_a \lozenge \pi>]\!] &&= \cup \{ SU[\![\pi_j]\!] \mid x^i_j \in SU[\![\pi]\!] \} \end{aligned}$$

The last equation is the most interesting: variable z may share with value v returned by the function call if there is a parameter x^i_j that shares with v and z shares with x^i_j.

It is not clear whether a good sharing analysis can really be built on this approach. The set of path actions suggested here is probably too primitive and small, so the sharing analysis is likely to be very conservative.

The way the interference analysis is extended to work for higher order functions cannot be used for this sharing analysis, so it applies only to first order functions.

8.2.2 Relative Globalization

Globalization of a variable means that it is replaced by a completely global variable that can be statically allocated. This was the only meaningful possibility in the languages L and H because these languages do not have nested scopes.

In languages with nested scopes (like Pascal or Scheme), one might define *relative globalization* of a variable. By this we mean that the variable may be made global in some scope which need not be the outermost (most global) one. The variable cannot be allocated statically then, but if scopes are entered and left in a first-entered-last-left manner (as in Pascal), then the variable may be allocated on evaluation stack at entry to the scope. The benefits of this are the same as for globalization: fewer copies of the variable need be kept at run-time.

For this relative globalization to be correct, the lifetime of the variable must be at least as long as the lifetime of the scope in which it has been made global.

It is likely that a slight extension of the path semantics could be used to detect cases where relative globalization can be done, for the nested call structures show the sequence of entries and exits of scopes during evaluation. Therefore it may suffice to add name tags to the call structures, so that the different lexical scopes can be distinguished in the paths. Then a more complex interference analysis may be able to find the most global (that is, the outermost) scope in which a variable can validly be made global.

8.2.3 Stackability

Stackability generalizes globalization: when can a heap for storage of dynamically allocated objects (such as function closures) be replaced by a (usual last-in first-out) stack? That is, how can one detect that (some) objects may be deallocated in the reverse order of that in which they were allocated? A good solution of this problem would be very useful in the implementation of higher order languages such as Scheme or Standard ML.

This is because heap allocation is more expensive in terms of run-time and storage than stack allocation. Particularly annoying is the need for garbage collection and the run-time overhead it incurs at unpredictable intervals. However, in extreme cases, heap allocation may be more efficient than stack allocation (in terms of run time, but then not in terms of storage). In theory, the asymptotic cost of garbage collection may be smaller than that of stacking and unstacking, when the amount of available storage is much greater than the amount of active cells at any time. With a two-space stop and copy garbage collector only the (few) cells actually in use need be visited and copied during garbage collection. Experiments confirm this [Appel 1987]. We shall assume, however, that stackability is desirable. Stackability has been studied in connection with attribute evaluators [Kastens 1987] and denotational language definitions [Schmidt 1986a].

This problem is more general than the problem of replacing stack usage by global variables, for if a variable can be allocated globally, then it can also be allocated in a stack (which will never contain more than one element). An analysis to detect stack-like allocation behaviour would therefore be more complicated, posing weaker requirements on a given program for (some of) its heap usage to be replaceable by stack usage.

The reason for allocating an object on the heap is that its lifetime cannot (easily) be predicted. This holds for closures as well as for cons cells in a Lisp system, say. Both may be returned by a function and hence survive the environment in which they were created. This means that they cannot be deallocated when the environment is left and its bindings are deallocated. Since the environment's bindings were allocated before the closure (say), it follows that allocation and deallocation are not done in a stack-like manner (in which what is last allocated should be first deallocated). The problem is even more acute because the closure value may be enclosed in other closures (similarly, cons cells may be made part of structures referred to by other cons cells).

It is likely that some variant of the path concept is useful for the detection of closures that may in fact be allocated on stack instead of heap, and similarly for cons cells. The requirement is that (apart from being applied) the closure is only passed as parameter to functions called from the environment in which the closure was created; it must not be returned as a result so that it survives the environment. This is known as a downward funarg in the Lisp community. The function call structures in paths give information about the order of creation and deletion of environments and thus should be suitable for checking this requirement. In addition it is necessary to have sharing information that can ensure that the closure is not enclosed in any other closure which may survive the environment. This information could possibly be delivered by a more sophisticated version of the sharing analysis sketched in Section 8.2.1.

8.3 Application to Lazy Languages?

Because of the present surge in the activity on lazy languages and their implementations, it would be desirable to apply our (operational) path semantics approach to the analysis of such languages.

It is rather straightforward to design an operational semantics for a version of L with *call by name*, and also the corresponding path semantics (that records every use of a variable). This form of path semantics can be used for strictness analysis: function f^i is strict in parameter position j if there is a use of x^i_j in *every* possible evaluation path for the body of f^i.

To obtain a version of L with *call by need* (or lazy evaluation), we may follow [Bloss, Hudak 1988] and replace the simple concatenation $\pi_1\pi_2$ in the (call by name) path semantics rules by an operation join(π_1,π_2) that removes all those uses $\uparrow x$ from π_2 that are already in π_1. Thus the path for evaluation of a function body will contain at most one use of every variable. This reflects the property of call by need that an argument expression is evaluated the first time its value is needed and nevermore. This kind of path semantics is not amenable to approximation by a context free grammar, however; the path for a composite expression is not a concatenation of the paths for its subexpressions.

9. CONCLUSION

We addressed the *globalization problem* for applicative languages: when may function parameters (local variables) be replaced by global variables? One practical motivation for this is the desire to reduce the run-time and storage claims of applicative programs. A related motivation is the applications in semantics-directed compiler generation: the automatic construction of efficient implementations from formal language definitions.

The criteria for globalization are *non-interference* and *non-enclosure* (for higher order languages; the first suffices for first order languages). We focused on the concept of interference and developed a simple theoretical framework for discussing interference in strict applicative programs; we introduced the concepts of definition-use path, path semantics, interference, and variable groups. The problems, concepts, and techniques were illustrated in detail using simple first order and higher order recursion equation languages.

We developed an automatic interference analysis and an automatic globalization transformation. A definition-use grammar construction, a variable grouping analysis, and a closure analysis were introduced to support them. We gave algorithms for all these constructions, but none of them has been implemented.

Correctness of the analyses and transformations has been emphasized throughout, and proofs have been given or sketched for all constructions.

We compared our globalization approach to other work with related goals or techniques. We found that our techniques are at least as powerful and theoretically well-founded as others known to us. Our techniques may turn out to be more complicated and more expensive to implement, however. While the techniques look promising, it is not clear whether the run-time and storage savings will make up for the effort required to apply them. Would it be reasonable to build these techniques into a compiler for applicative languages? An implementation of the analyses and transformations would cast more light on these questions.

Finally, several possible extensions and improvements of the approach were discussed. A particularly useful extension would allow us to detect cases where heap usage could be replaced by stack usage. This would allow us to detect automatically those parts of a higher order language that might be implemented with stack (*e.g.*, for storing closures) instead of heap.

Another direction is the possible extension of this approach to languages that do not have strict application, such as lazy languages.

10. GLOSSARY OF SYMBOLS

General

□	end of Example, Definition, Algorithm, Proposition, or Lemma

Sets and Functions

A, B, I	sets
$\wp(A)$	the powerset of A, *i.e.*, the set of all subsets of A
card(A)	the cardinality of A, *i.e.*, its number of elements (for finite A only)
$\cup A$	distributed union of a family A of sets, *i.e.*, $\{ b \mid \exists a \in A.\ b \in a \}$
$\{ a_i \}_{i \in I}$	the set of a_i's indexed by $i \in I$, *i.e.*, the set $\{ a_i \mid i \in I \}$
$h: A \to B$	the set of total functions from A to B
$h: A \dashrightarrow B$	the set of partial functions from A to B
$\mathrm{dom}(h: A \dashrightarrow B)$	the domain of h, *i.e.*, the set of $a \in A$ for which h(a) is defined
$\mathrm{rng}(h: A \dashrightarrow B)$	the range of h, *i.e.*, the set of $b \in B$ for which $\exists a \in A.\ h(a)=b$

The Language L (Section 2.2)

pgm	a given program
e	an expression
e^0	the initial expression in a given program
A	a basic function
f^i	a defined function, $i \in I$
arity(f^i)	the arity of f^i, *i.e.*, its number of parameters (local variables)
$x^i_j \in$ Var	local variable, *i.e.*, parameter position j of function f^i
$X \in$ GloVar	global variable
$v \in$ Value	a value
FreeVars(e)	the set of variables with (free) occurrences in e
Input = FreeVars(e^0)	the set of input parameters in a given program

The Language H (Section 4.1)

$f^i(v_1,\ldots,v_m) \in$ Closure	a closure, *i.e.*, a functional value (where $m <$ arity(f^i))
$\xi \in$ Var $\cup$ GloVar	a local or global variable in the parameter list of an imperative H program

Operational Semantics (Sections 2.2 and 4.1)

$\rho \in$ Env = Var $\dashrightarrow$ Value	a local environment
$\rho_0 \in$ Input $\dashrightarrow$ Value	the input to a given program, *i.e.*, values for its input parameters
$\sigma \in$ State = GloVar $\dashrightarrow$ Value	a global state
$\rho \vdash e \Rightarrow v$	in environment ρ, expression e may evaluate to value v
$\rho, \sigma_1 \vdash e \Rightarrow v, \sigma_2$	in ρ and initial state σ_1, e may evaluate to v with final state σ_2

Paths and Path Semantics (Sections 2.3 and 4.2)

$\uparrow x \in$ Use	use of x
$\downarrow x \in$ Def	definition of x
$y»x \in$ Copy $\subseteq$ Def	copy, *i.e.*, definition of x by a copying of y's value
$\pi \in$ Path	a du-path
ε	the empty path (length zero)
$\delta \in$ Def	a definition or a copy
$< \pi_1\delta_1 \ldots \pi_a\delta_a \Diamond \pi >$	a call structure with prelude $\pi_1\delta_1 \ldots \pi_a\delta_a$ and body π
$\rho \vdash e \Rightarrow v, \pi$	in environment ρ, e may evaluate to v and trace path π
$\Pi(e)$	the set of paths possible for expression e
$\Pi(pgm)$	the set of paths possible for program pgm
$\Delta(x^i_j,e)$	is $y»x$ if e is an occurrence of variable y, $\downarrow x$ otherwise

Interference and Variable Groups (Section 2.4)

$U_0(\pi)$	the set of variables with level 0 uses in π
$D(\pi)$	the set of variables with definitions in π
Interf(π) $\subseteq$ Var	the interference of path π
$\gamma \in$ VGroup	a variable group, *i.e.*, a non-empty set of variables
$\Gamma \in$ VGrouping	a variable grouping, *i.e.*, a set of disjoint variable groups
$DG(\pi)\Gamma$	the set of variables with definitions in π, relative to Γ
Interf(π,Γ)	the interference of path π relative to Γ
Interf(pgm,Γ)	the interference of program pgm relative to Γ
$x_\gamma \in$ GloVar	the global variable replacing all variables in variable group γ

Definition-Use Grammars (Section 3.1)

G_{pgm}	the du-grammar for a given program
$N \in V_N$	a nonterminal symbol
N_e	the du-grammar nonterminal corresponding to expression e
N_{pgm}	the du-grammar start nonterminal
$\alpha \in$ Rhs	a grammar rule right hand side, *i.e.*, a sequence of grammar symbols
$\alpha \rightarrow^* \pi$	path π is derivable from α (in particular for α being a nonterminal N_e)

Interference Analysis on Grammars (Section 3.2)

$ua_0(\alpha)$	superset of the set of variable uses in all π with $\alpha \rightarrow^* \pi$
$uenv \in UEnv$	superset of the set of variable uses in all π with $N \rightarrow^* \pi$
$UA_0[\![\alpha]\!]uenv$	superset of the set of variable uses in all π with $\alpha \rightarrow^* \pi$
$da_\Gamma(\alpha)$	superset of the set of variable definitions in all π with $\alpha \rightarrow^* \pi$
$denv_\Gamma \in DEnv$	superset of the set of variable definitions in all π with $N \rightarrow^* \pi$
$DA[\![\alpha]\!]\Gamma\ denv_\Gamma$	superset of the set of variable definitions in all π with $\alpha \rightarrow^* \pi$
$\S = ia(G_{pgm})\Gamma$	safe approximation to Interf(pgm,Γ), *i.e.*, $\S \supseteq$ Interf(pgm,Γ)
$ienv_\Gamma \in IEnv$	$ienv_\Gamma(N)$ approximates Interf(π,Γ) for all π with $N \rightarrow^* \pi$
$IA[\![\alpha]\!]\Gamma\ ienv_\Gamma$	approximation to Interf(π,Γ) for all π with $\alpha \rightarrow^* \pi$

Non-Interfering Variable Groupings (Section 3.3)

$\#\Gamma$	the quality index for variable grouping Γ
(V, E)	the directed graph with vertex set V and edge set $E \subseteq V^2$
$p: V \rightarrow Q$	a graph colouring, *i.e.*, p is surjective, and $p(a) = p(b)$ implies $(a,b) \notin E$
Γ_{non}	the non-interfering variable grouping constructed for a given program

The Globalization Transformations (Sections 3.4 and 4.5)

pgm_Γ	pgm with each variable group in Γ replaced by a global variable
$T[\![e]\!]\Gamma$	the result of globalizing the variable groups of Γ in expression e

The Approximate Closure Analysis (Section 4.3)

$ca(e)$	the abstract closures that evaluation of e may return
$c: AClo$	an abstract closure (f^i,m) with $0 \le m < arity(f^i)$ and $i \in I$
$cf: CF=I\rightarrow \wp(AClo)$	$cf(f)$ is the set of abstract closures that a call to f^i may return
$CA[\![e]\!]cf\ cv$	the abstract closures that evaluation of e may return
$cv: CV=Var\rightarrow \wp(AClo)$	$cv(x)$ is the set of abstract closures to which x may be bound
$VA[\![e]\!](k,j)\ cf\ cv$	the abstract closures that evaluation of e may bind to variable x^k_j

11. REFERENCES

[Aho, Hopcroft, Ullman 1974]
A.V. Aho, J.E. Hopcroft and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley 1974.

[Aho, Sethi, Ullman 1986]
A.V. Aho, R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley 1986.

[Appel 1987]
A.W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters* **25** (1987) 275-279.

[Backus 1978]
J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* **21**, 8 (August 1978) 613-641.

[Bird 1976]
R. Bird. *Programs and Machines. An Introduction to the Theory of Computation.* John Wiley and Sons 1976.

[Bloss, Hudak 1988]
A. Bloss and P. Hudak. Path semantics. In M. Main *et al.* (eds.): *Mathematical Foundations of Programming Language Semantics, 3rd Workshop*, New Orleans, Louisiana, April 1987. *Lecture Notes in Computer Science* **298** (1988) 476-489. Springer-Verlag.

[Brélaz 1979]
D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM* **22**, 4 (April 1979) 251-256.

[Ershov 1978]
A.P. Ershov. On the essence of compilation. In E.J. Neuhold (ed.): *Formal Description of Programming Concepts*, 391-420. North-Holland 1978.

[Farrow, Yellin 1986]
R. Farrow and D. Yellin. A comparison of storage optimizations in automatically-generated attribute evaluators. *Acta Informatica* **23** (1986) 393-427.

[Futamura 1971]
Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls* **2**, 5 (1971) 45-50.

[Garey, Johnson 1979]
M.R. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company 1979.

[Gill 1965]
S. Gill. Automatic computing: its problems and prizes. *Computer Journal* **8**, 3 (1965) 177-189.

[Haskell 1975]
R. Haskell. Efficient implementation of a class of recursively defined functions. *Computer Journal* **18**, 1 (1975) 23-29.

[Henderson 1980]
P. Henderson. *Functional Programming. Application and Implementation.* Prentice-Hall International 1980.

[Hoare 1972]
C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica* **1** (1972) 271-281.

[Hughes 1982]
J. Hughes. Supercombinators: a new implementation method for applicative languages. *1982 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, Pennsylvania, 1-10.

[Jensen, Dam 1985]
F. Jensen and M. Dam. Automatic Compiler Generation from Operational Semantics Definitions of Programming Languages. M. Sc. thesis, 141 pages. Aalborg University Centre, Denmark 1985 (in Danish).

[Johnsson 1985]
T. Johnsson. Lambda lifting: transforming programs to recursive equations. In J.-P. Jouannaud (ed.): *Functional Programming Languages and Computer Architecture*, Nancy, France, 1985. *Lecture Notes in Computer Science* **201** (1985) 190-203. Springer-Verlag.

[Jones 1987]
N.D. Jones. Flow analysis of lazy higher-order functional programs. In S. Abramsky and C. Hankin (eds.): *Abstract Interpretation of Declarative Languages*, 103-122. Ellis Horwood, Chichester, England 1987.

[Jones, Sestoft, Søndergaard 1988]
N.D. Jones, P. Sestoft and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation* **1**, 3/4 (1988) (to appear). Kluwer Academic Publishers.
Also DIKU Report 87/8, 52 pages, DIKU, University of Copenhagen, Denmark 1987.

[Kahn 1987]
G. Kahn. Natural Semantics. Rapport de Recherche 601, 22 pages, INRIA, Sophia-Antipolis, France 1987.

[Kastens 1987]
U. Kastens. Lifetime analysis for attributes. *Acta Informatica* **24** (1987) 633-651.

[Kastens, Hutt, Zimmermann 1982]
U. Kastens, B. Hutt and E. Zimmermann. *Gag: A Practical Compiler Generator. Lecture Notes in Computer Science* **141** (1982) 156 pages. Springer-Verlag.

[Kastens, Schmidt 1986]
U. Kastens and M. Schmidt. Lifetime analysis for procedure parameters. In B. Robinet and R. Wilhelm (eds.): *ESOP 86. European Symposium on Programming*, Saarbrücken, Federal Republic of Germany, March 1986. *Lecture Notes in Computer Science* **213** (1986) 53-69. Springer-Verlag.

[Knuth 1974]
D.E. Knuth. Structured programming with go to statements. *Computing Surveys* **6**, 4 (December 1974) 261-301.

[Lyndon 1966]
R.C. Lyndon. *Notes on Logic.* D. Van Nostrand Company 1966.

[Mosses 1979]
P.D. Mosses. SIS - Semantics Implementation System, Reference Manual and User Guide. DAIMI Report MD-30, DAIMI, Aarhus University, Denmark 1979.

[Mycroft 1981]
A. Mycroft. Abstract Interpretation and Optimising Transformations for Applicative Programs. Ph. D. Thesis, Report CST-15-81, 180 pages, Department of Computer Science, University of Edinburgh, Scotland 1981.

[Pettorossi 1978]
A. Pettorossi. Improving memory utilization in transforming recursive programs. In J. Winkowski (ed.): *Mathematical Foundations of Computer Science 1978. 7th Symposium*, Zakopane, Poland. *Lecture Notes in Computer Science* **64** (1978) 416-425. Springer-Verlag.

[Pettorossi 1984]
A. Pettorossi. Constructing recursive programs which are memory efficient. In A.W. Biermann, G. Guiho and Y. Kodratoff (eds.): *Automatic Program Construction Techniques*, 289-303. Macmillan Publishing Company, New York 1984.

[Peyton Jones 1987]
S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International 1987.

[Plotkin 1981]
G.D. Plotkin. A Structural Approach to Operational Semantics. DAIMI Report FN-19, 172 pages, DAIMI, Aarhus University, Denmark 1981.

[Reynolds 1978]
J.C. Reynolds. Syntactic control of interference. *Fifth ACM Symposium on Principles of Programming Languages*, Tucson, Arizona 1978, 39-46.

[Schmidt 1985]
D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems* **7**, 2 (April 1985) 299-310.

[Schmidt 1986a]
D.A. Schmidt. Detecting Stack-Based Environments in Denotational Definitions. (Extended Version). Report TR-CS-86-3, 36 pages, Department of Computing & Information Sciences, Kansas State University, Kansas 1986.

[Schmidt 1986b]
D.A. Schmidt. *Denotational Semantics. A Methodology for Language Development.* Allyn and Bacon, 1986.

[Schmidt 1988]
D.A. Schmidt. Static properties of partial evaluation. In D. Bjørner, A.P. Ershov and N.D. Jones (eds.): *Partial Evaluation and Mixed Computation*, 465-483. North-Holland 1988 (to appear).

[Steele 1977]
G.L. Steele Jr. Debunking the "expensive procedure call" myth. *ACM Annual Conference*, Seattle, Washington, October 1977, 153-162.
Also AI Memo 443, 22 pages, MIT, Cambridge, Massachusetts 1977.

[Steele 1978]
G.L. Steele Jr. Rabbit: A Compiler for Scheme (A Study in Compiler Optimization). Technical Report AI-TR-474, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts 1978.

[Stoy 1977]
J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press 1977.

[Strong 1971]
H.R. Strong. Translating recursion equations into flow charts. *Journal of Computer and System Sciences* 5 (1971) 254-285.

[Tofte 1984]
M. Tofte. *Compiler Generators - What They Can Do, What They Might Do, and What They Will Probably Never Do. EATCS Monographs on Theoretical Computer Science.* Springer-Verlag 1988 (to appear).
Also DIKU Report 84/8, 224 pages, DIKU, University of Copenhagen, Denmark 1984.

[Turner 1982]
D.A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson and D.A. Turner (eds.): *Functional Programming and Its Applications. An Advanced Course*, 1-28. Cambridge University Press 1982.

[Walker, Strong 1973]
S.A. Walker and H.R. Strong. Characterizations of flowchartable recursions. *Journal of Computer and System Sciences* 7 (1973) 404-447.

[Wulf *et al.* 1975]
W. Wulf *et al. The Design of an Optimizing Compiler*. American Elsevier Publishing Company, New York 1975.