PARAMETER SPLITTING IN A HIGHER ORDER FUNCTIONAL PROGRAMMING LANGUAGE

Bjarne Steensgaard [1] Morten Marquard [2]

DIKU
Department of Computer Science
University of Copenhagen
Universitetsparken 1
2100 København Ø.

August 7, 1990

[1]e-mail: rusa@diku.dk
[2]e-mail: marquard@diku.dk

Variable Splitting in a Higher Order Functional Language

This project is a part of a larger one consisting of constructing a Miranda to 'C' compiler. This compiler is to be constructed by first implementing a Miranda-interpreter in Scheme, then using a partial evaluator to generate a Miranda to Scheme compiler written in Scheme, and finally generating 'C'-code from the Scheme-code.

We will develop a theory for doing parameter splitting in a higher order functional programming language based on a theory for doing parameter splitting in a first order functional programming language [Romanenko 90].

A reason for doing parameter splitting in connection with program specialization is given the abstract of [Romanenko 90]. It states:

> Experiments on generating compilers by specializing specializers with respect to interpreters have shown that the compilers thus obtained have a natural structure only if the specializer does *variable splitting*. Variable splitting can result in a residual program using several variables to represent the values of a single variable of the original program. In the case of functional programming, variable splitting is done be raising the arities of functions.

The language to be transformed is to be a subset of Scheme.

We intend to implement a parameter splitting program based on the theory we develop. The program will be implemented to work in an environment with Anders Bondorf's Similix-2 [Bondorf 90].

Instead of only handling the constructor `cons` and the selectors `car` and `cdr` we want the parameter splitting program to be able to handle (abstract) datastructures like Similix-2 does. The theory we are going to develop will of course reflect this.

We will describe the theory in a paper. This is done in the hope that the paper might be published some time. The paper will be the main result of this project, but we will nevertheless describe the developed programs in a report.

Bjarne Steensgaard-Madsen & Morten Marquard, July 1'st, 1990.

References

[Bondorf 90] Bondorf, A. Automatic Autoprojection of Higher Order Recursive Equations. In Jones, Neil D. (editor), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 70–87. Springer-Verlag, May 1990.

[Romanenko 90] Romanenko, Sergei A. Arity Raiser and Its Use in Program Specialization. In Jones, N. (editor), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Springer-Verlag, 1990.

Contents

Chapter 1

Introduction

This report has been produced during the authors' work with parameter splitting in a higher order functional language during the spring semester of 1990. The work originates in a groups attempts to implement a Miranda to C compiler. This should be done by implementing a Miranda interpreter in Scheme that can be specialized with the aid of a automatic program specializer, *Similix-2* [Bondorf 90]. The specializer can transform the Miranda interpreter to a Miranda to Scheme compiler. With the addition of a Scheme to C compiler we should in effect have a Miranda to C compiler.

When compiling programs with a compiler produced in this way from an interpreter, the resulting program often has an unnatural structure. Many values are kept in lists or other composite datastructures. With a parameter splitting program these composite datastructures can be split into several pieces. This often improves the readability of the program and makes certain local optimizations possible. Furthermore the parameter splitting process can in some cases make certain optimizations possible that are connected with partial static datastructures.

In [Romanenko 90] an algorithm is given that can be used to do parameter splitting in a first order functional language. We have generalized the theory to handle higher order constructs in a strict functional programming language. Furthermore we have generalized the theory to handle (abstract) datastructures in general rather than only handle the constructor `cons` and the selectors `car` and `cdr`. We have then written and implemented a parameter splitting program that handles the subset of Scheme used in *Similix-2*. This subset is a higher order language.

During the work with generalizing the theory for doing parameter splitting we produced a paper describing the results. This paper should be regarded as a part of this report. The report also includes a description of the developed programs, gives a description of how they have been tested and gives a description of how to use them.

The authors regard the paper as the main result of this project. We have worked quite a lot with the paper in order to give it a reasonable quality. The rest of the report is not by the authors regarded as important as the paper and has not been subject to the same amount of work.

Chapter 2

The theory behind parameter splitting

We have described our work with the theory behind parameter splitting in a paper. This paper appears as appendix A. In the paper we describe an algorithm to perform parameter splitting in a higher order functional language. How to perform parameter splitting in a first order functional language is described in [Romanenko 90].

The theory is described in an article style paper in order to submit it for publication sometime in the future. The subject and the theory should be interesting enough to be published.

Most of the text in the paper could of course be reformatted and included in the main part of this report. We have chosen not to do so. This is because we believe that we can put our time to better use than trying to rephrase certain parts of the paper in order to make them fit better into this report.

The paper is still regarded as the main result of our work.

The closure-analysis described in the paper treats closures as part of a composite datastructure better than all other closure-analysises we have seen. Yet, it can be improved by retaining some closure and structure information when generalizing the type of a variable or when finding the least upper bound of a set of variables. We haven't done this since it might lead to even slower execution times of the parameter splitting program.

During the implementation of the algorithms we have also done some experiments with lambda lifting. The analysises that are performed give all the information needed to perform lambda lifting.

We haven't described the theory behind lambda lifting anywhere. We have just played with some ideas and tried to implement them as part of the parameter splitting program. It seems to be working correctly even though we haven't tried working out a formal theory.

It is very probable that we will write something about lambda lifting in the near future.

Chapter 3

How to use the program

Using the parameter splitting program is very simple. Any program that can be or has been specialized by *Similix-2* can be transformed by the parameter splitting program simply by issuing the following command:

(ps "*file-name*")

The transformed version of the program will be written into a file with the same name as the original one, except that the prefix "ps-" has been added to the name.

3.1 What can the program do?

The program can of course perform parameter splitting as described in the paper (functions returned as (part of) the result of the goal function may be split). But it can also do more than that.

As mentioned in our paper, it is possible to add a lambda lifting routine to the parameter splitting program. We have in fact done this. There is a flag that can be used to control whether or not the parameter splitting program shall perform lambda lifting as part of the transformation.

There is another flag that controls the amount of output the parameter splitting program shall provide while transforming a program. When the flag is set, the program outputs timing information for each phase of the transformation. During the splitting phase it also outputs information on what parameters are being modified. No attempts to make the output readable has been done.

In order to be truly compatible with the *Similix-2* program specializer, the programs to be transformed may include `load` statements and syntactic extensions.

The `load` statements may only appear at the outmost level. The files to be loaded may include other `loadt` commands, syntactic extensions, function definitions and other `load` statements. The goal-function is still the first function to appear in the original file.

The syntactic extensions used in *Similix-2* resemble those of Chez-Scheme very much. They are limited in the way, that they don't have the ... notation, doesn't accept additional keywords, doesn't accept a *fender* and doesn't do pattern matching. The routine for doing syntactic extensions in the parameter splitting program is more general. It works

exactly like the Chez-Scheme syntactic extensions. The parameter splitting program of course accepts those syntactic extensions that *Similix-2* does. A set of standard syntactic definitions is read from the file `extend-syntax.ss`.

The parameter splitting program can thus handle more programs than *Similix-2* can. There are still many programs that the parameter splitting program might not treat correctly. If any functions has side-effects in the form of `set!`, `set-car!` or `set-cdr!` the parameter splitting program may produce incorrect results.

One consequence of the missing capability to handle programs with side-effects is that the parameter splitting program doesn't accept programs with definitions of the form:

(`define` *name exp*)

at the outmost level. In *Chez-Scheme* this will be expanded to a `set!` expression. We haven't defined that syntactic extension since the parameter splitting program doesn't treat side-effects in a correct manner. `define` statements at the outmost level defining functions are dealt with specifically and are thus not a problem.

It is possible to define a syntactic extension that allows definitions (with `define`) of functions and non-functions within other functions, but we haven't done this. It can be done by modifying the syntactic extension for `begin`.

Apart from the programs that have side-effects (directly or indirectly), the parameter splitting program can, as far as we can see, (potentially) transform all programs that are written in *Chez-Scheme*. Some constructs hasn't been defined yet by *extend-syntax* but that is just a question of modifying the file `extend-syntax.ss`.

3.2 Using the different features

As already mentioned the parameter splitting program can transform **any** program that can be or has been specialized by *Similix-2*. This is done simply by issuing the command:

(`ps "`*file-name*`"`)

To manipulate the flag that controls whether or not the program shall do lambda lifting in the program being transformed we have defined three functions:

```
(rll)          ; raise the flag - turn lifting on
(cll)          ; clear the flag - turn lifting off
(pll)          ; print the current value of the flag
```

We have also defined three functions to manipulate the flag that controls whether the parameter splitting program shall give timing information or not:

```
(rtime)          ; raise the flag - turn output on
(ctime)          ; clear the flag - turn output off
(ptime)          ; print the current value of the flag
```

The parameter splitting program makes use of the `adt`-files that are loaded with the `loadt` statement. The program can use almost the same `adt`-files as *Similix-2* does. The only restriction is that there must only be one constructor associated with each sort. If there is more than one constructor definition in a sort definition an error message will be issued.

In order to have the parameter splitting program perform any parameter splitting the `adt`-files must be changed in one other way. For each sort supposed to be split, there has to be a rule defining the relationship between the constructor and the selectors in the sort. The rule consist of a list of selector names accessing the different components of the datastructure built by the constructor. The order in the list must reflect the components accessed. This extra information is added as a fifth field in the `define-constructor` part of the `define-sort` construction. For the sort `pair` in the file `scheme.adt` (taken from *Similix-2*) the definition of the constructor `cons` becomes:

```
(define-constructor
   cons
   ((any any) pair)
   transparent
   cons  (car cdr))
```

This modification does not interfere with *Similix-2* in any way.

We have further added the possibility of giving a `define-predicate-is-sort` definition in the `define-sort` construction. The predicates defined in this way must return *true* if its argument is an element of the corresponding sort and *false* otherwise. This modification is not absolutely necessary, but it can lead to better results if the program to be transformed contain constant data. The disadvantage is that *Similix-2* doesn't accept this kind of definition.

Chapter 4

Description of the program

We have implemented a fully automatic parameter splitting program handling the subset of Scheme used by the specializer, *Similix-2*. We have retained the idea of using `adt`-files to specify which selectors, constructors, predicates and operators are to be included in the treated language. In this chapter we will describe the program that we have produced.

The programs total approximately 4000 lines of Scheme code and is larger in size than *Similix-2*. The description will therefore not be of details in the program but rather of the general ideas. The theory behind the program is described in chapter 2 and in appendix A.

The program consists of 5 almost independent subprograms and some functions tying the whole thing together. The program is listed in a total of 8 files. We will describe what the different files contain and what actions the different subprograms perform. A program listing is given in appendix B.

One important issue is that we first generate a new program with an abstract syntax from the program to be transformed. The abstract syntax contains several extra fields supposed to contain extra information. The subprograms then work on this abstract syntax, destructively updating different fields of the abstract syntax of the program. When the analysis and program transformation have been performed on the program with the abstract syntax, then the program is transformed to a program with a Scheme syntax.

The program is written in the same style as the *Similix-2* program is. Anyone who has read the *Similix-2* program text shouldn't have any trouble reading ours. Everybody who hasn't read the *Similix-2* program text will probably find the program difficult to read.

There are only a few places in the program where we have used dirty programming. Most of the program is written in such a manner that we can defend giving it to other people. The program is intentionally not commented in a large degree. To understand the program, you have to understand the theory. If you understand the theory in full, then you should have no trouble understanding most of the program (the lambda lifting parts excluded).

4.1 The abstract syntax (abstract.ss)

We have defined an abstract syntax that most of the subprograms work with. We have defined several predicates, selectors, modifiers and lookup-functions working on this abstract syntax. All these functions are listed in the file `abstract.ss`.

In the top of the file we have given the actual syntax of the lists representing the program to be transformed. This is to aid the reading of the functions listed below in the file.

The program is everywhere reachable because it is given in lists pointed to by global variables. All user defined procedures (udp) are given in the list `**vs-udp***`. All user defined nameless lambda abstractions (uda) and let-statements are given in the lists `**vs-uda***` and `**vs-let***`. All applications are given in the list `**vs-app***`. The elements of the list are shared whenever possible. That means that if a part of one of the elements in a list is modified, then the corresponding part (if any) in the other lists is also changed.

In the analysis phases of the parameter splitting process, type and context information is stored in the abstract program. All functions used to build and retrieve this information is also defined in the file `abstract.ss`. The actual syntax for the stored information is listed just before the functions used to build and access the information.

The information contained in loaded `adt`-files is also kept in abstract datastructures. Functions used for obtaining information from these abstract datastructures are listed at the end of the file. All needed information about predicates and operators can be found in the lists `**vs-Predicate***` and `**vs-Operator***`. All needed information about constructors and selectors can be found in the list `**-vs-csd***`.

For practical reasons we have separated basic (or "builtin") sorts and all other sorts. All information about basic sorts can be found in the list `**vs-isBasicSort***`. The list `**vs-isNormalSort***` contains information about all the other sorts. All the information from the loaded `adt`-files are kept in the list `**vs-AdtFile***`.

4.2 Generating the new program (pre.ss)

The Preprocessing of a program is done in two steps. First the program is macro expanded according to the rules given by the `extend-syntax` clauses in the program and in the file `extend-syntax.ss`. Then the program with the abstract syntax as defined in `abstract.ss` is generated from the macro expanded program.

In the top of the file we have given both the concrete syntax before and after macro expansion of the lists representing the program to be transformed. This is to aid the reading of the functions listed later in the file.

The macro expansion of the concrete syntax is done according to the rules given by the `extend-syntax` clauses of the program plus the clauses found in the file `extend-syntax.ss`. This is to make our syntax as similar to both *Chez-Scheme*'s and *Similix-2*'s. Our extend-syntax mechanism is fully compatible with the mechanism given in *Chez-Scheme*. Given a clause, it first tries to match the clause's pattern to a given expression, binding variables in the pattern to its corresponding subparts in the expression. If the match succeeds,

the pattern matching function returns an environment, i.e. an *association list*. Then the *fender* (if any) is evaluated. If the evaluation of the fender returns false the clause does not match the expression. If it succeeds, the clauses' expansion is returned, with all occurrences of variables in the expansion bound to the values given in the environment. If the expansion contains a with-expression, then the environment is augmented with the values found by evaluating the with-expression (i.e. binding values to names) and then the expansion in the with-expression, is expanded according to the new environment.

The macro expansion part of the preprocessor is easy to move to *Similix-2*. We have already talked with Anders Bondorf (one of the authors of *Similix-2*) about it and he seems *very* interested.

The generating of the program with the abstract syntax is straightforward. All user defined procedures (udp) are converted to equivalent procedures with the abstract syntax and added to the list `**vs-udp***`. All user defined abstractions (uda) are added to the list `**vs-uda***`, all let-expression to the list `**vs-let***` and finally all applications to the list `**vs-app***`. All fields in the abstract syntax that is supposed to have an initial value is also set during the generation of the program with the abstract syntax. As mentioned before, the elements of the lists is shared whenever possible.

There is some superfluous functions defined in the file `pre.ss`. Most of them are relics from the time when the program didn't support syntactic extensions.

4.3 Performing the forward analysis (forward.ss)

In our paper we have described the functions `B` and `D` that are used to perform the forward analysis. The two functions are implemented almost as described in the paper.

Instead of updating an argument type description that is passed as parameter between function calls, the function `D` updates fields in the abstract data-structure. This is of course done with the functions implementing the abstract syntax of the treated program. The environment is represented by the name of the lambda abstraction that the expression is part of. By using the lookup functions defined in `abstract.ss` it is possible to retrieve all the information supposed to be in the environment and in the argument type description.

The functions used to implement the least upper bound operation is listed in the top of the file. The least upper bound is not defined exactly as in the article. This is because we have taken *number*, *lambda* and *atom* to be sortdescriptors with one argument.

The functions used to control closure-sets and closure-super-sets are defined in the bottom of the file. Most of the complicated work related to ensuring that all lambda abstractions in the closure-super-set have the same argument types is done in the function `_vs-maintain-monovariant-invariant`.

4.4 Performing the backward analysis (backward.ss)

In our paper we have described the functions *TypeToContext*, *GenType* and `C` used to perform the backward analysis. The three functions are implemented as described in the

paper. The three function are fully applicative, i.e. no updating is done destructively in the abstract syntax.

Further, we have implemented a routine, that iteratively finds the contexts of parameters of all functions, abstractions and let-expressions, and destructively updates the abstract syntax, until a fixed point is found.

For technical reasons we have to start by calculating the types of the parameters at each application point. These types are taken from the result type description from the forward analysis based on information of which functions may be applied at the application point in focus. The types from the forward analysis is changed to the types in the paper. This is done in the function `_vs-calculate-application-types!`.

To perform the iterations we need a first "guess" at all the contexts of parameters. These contexts are calculated in the function `_vs-find-context!`

Finally, having found the context of all parameters, these are generalized according to their type and context, and the abstract syntax is destructively updated.

4.5 Performing the parameter splitting (varsplit.ss)

In our paper we have described the functions CountGabs, ExpandVar and SplitArg that are used to perform the actual splitting of the program. The three function are implemented basicly as described in the paper.

The parameter splitting program has the ability to perform lambda lifting. The lifting of lambda abstractions is also performed by the functions given in this file. The parts that perform lambda lifting is intermingled with the other parts of the functions so the functions might be difficult to understand.

4.6 Generating the Scheme program (post.ss)

The postprocessing functions of the system uses the abstract syntax to generate concrete Scheme syntax. In an attempt to give more readable programs as output the functions try to find expanded `and`, `or` and `cond` expressions in the transformed program. If any such expressions are found they are transformed back into concrete `and`, `or` and `cond` expressions.

When generating the concrete syntax all names are changed in the following ways:

1. The names from the `adt`-files remain unchanged.

2. Functions (udp's) are given a new name, which equals the old name with the suffix "-global".

3. All other names (variable names) are changed. The new names equals the old names with a suffix consisting of the name of the procedure, lambda abstraction or let expression in which it is defined, an offset number that equals the offset in the list of formal parameters before variable splitting, and finally a number that equals the split parameters number in the list of new variables given as parameter

to *ExpandVar*. Given a variable `x` used as the second formal parameter in the definition of the function `f`, splitting this parameter will result into 2 new formal parameters with the names `x-f-1-1` and `x-f-1-2` in the residual program. Note that this way of renaming restricts the choice of names in the `adt`-files if it must be guaranteed the there will be no name clashes. The names of variables used in programs to be transformed is not restricted in any way.

4.7 Miscellaneous functions (misc.ss, system.ss and ps.ss)

The file `ps.ss` contain the function `ps` that calls all the parts of the parameter splitting system. It also contains all the functions needed to set or clear the flags controlling the different kinds of output.

The file `system.ss` is the file to load when you want to use the parameter splitting system. When it is loaded it loads all the other files in the system.

In the file `misc.ss` the miscellaneous functions not used by any part of the system in particular is listed.

Chapter 5

Tests of the programs

We have written a lot of small program that each test one or more properties of the parameter splitting program. In Appendix C we have listed all the source programs and the corresponding transformed programs. Each source program has been commented in order to show what property of the parameter splitting program that specific program test.

In our paper we have given several examples of programs to be split. We have tried transforming all theses examples in order to see if the transformation is performed as we expect. In appendix C we have also listed all the programs used as examples in the paper and the corresponding transformed programs.

We have also tried executing the parameter splitting program with *specializer* and *cogen* from *Similix-2* as input. We have also tried with a compiler generated by *Similix-2* as input. We refrain from giving input and output from these tests due to the size of the files (The size of the transformed version of *cogen* is > 400Kb). Instead we give timing information and information about how much is being split in appendix D.

The parameter splitting program seems to be working exactly as it should. At least we haven't been able to find any discrepancies between what we expected and what the actual results are.

Since we started working on this project the program specializer *Similix-2* has been changed so it does some kind of parameter splitting when self-applied. It does this by using a higher order environment instead of lists. When self-applied it can unfold these higher order environments and thus remove the list-accesses. Part of the goal of this project has therefore been solved by *Similix-2*.

Because *Similix-2* has been changed to do these reductions there is little use for a parameter splitting program in an environment with *Similix-2*. It is possible to perform some parameter splitting in the specializer, in some generated compilers and in *cogen*, but it is not much. In most cases where the parameter splitting program can do real work, the program is the result of a less than optimal design of the source program.

The parameter splitting program can still find good use in environments with specializers not using higher order environments. It can also improve a specializers treatment of programs with partially static datastructures. If any partially static datastructures occur in the program to be transformed, the parameter splitting program can split these into

several parts. The static and dynamic parts of the datastructure will thus be separated.

Chapter 6

Future work

The parameter splitting program is almost completely compatible with the program specializer *Similix-2*. The parameter splitting program can also do some things in a better way than *Similix-2* does. An obvious job is to make the two programs completely compatible with each other.

The parameter splitting program performs splitting in a monovariant way. In the future the program may be extended to perform splitting in a polyvariant way.

We might also want to occupy ourselves with splitting functions into two or more functions if the result of the original function is a composite datastructure. When Masami Hagiya from the Research Institute for Mathematical Sciences, Kyoto University, visited DIKU in may 1990 he asked us if that was possible. He wanted to use a result splitting program to extract programs from constructive proofs. We are both interested in the idea and in working on that project. We asked Sergei A. Romanenko if he knew something about the subject, but he told us that it is a whole new area that hasn't been investigated yet. This just makes it more interesting for us.

Chapter 7

Summary

We have generalized an algorithm used to perform parameter splitting in a first order functional programming language to an algorithm that can be used in a higher order functional language. This higher order language is the same as the source language of the *Similix-2* program specializer.

The theory concerning parameter splitting in a higher order language has been described in a paper. The theory involves a forward and a backward analysis of the program to be transformed. We are very satisfied with the report except for the first introductory section. We believe that we can't write a significantly better paper without receiving extensive constructive criticism.

We have designed, implemented and tested a program performing parameter splitting using the developed algorithm. The program can perform parameter splitting as described in the paper of all programs that can be or has been specialized by the program specializer *Similix-2*.

The program can also perform lambda lifting as part of the program transformation. The theory concerning lambda lifting isn't described in the paper or in this report. This is because we haven't developed a suitable formal description yet. We consider writing a paper about lambda lifting in the near future.

The program is also able to accept more advanced syntactic extensions than *Similix-2*. The syntactic extensions can be just as advanced as they can be in *Chez-Scheme*.

The program appears to work the way it should. At least we haven't been able to find any errors in the output from the examples we have tried. The examples we have tested the program which range from small test-program testing special parts of the program to programs as large as *cogen* from *Similix-2*.

Bibliography

[Bondorf 90] Bondorf, A. Automatic Autoprojection of Higher Order Recursive Equations. In Jones, Neil D. (editor), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 70–87. Springer-Verlag, May 1990.

[Romanenko 90] Romanenko, Sergei A. Arity Raiser and Its Use in Program Specialization. In Jones, N. (editor), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Springer-Verlag, 1990.

Appendix A

The paper

PARAMETER SPLITTING IN A HIGHER ORDER FUNCTIONAL PROGRAMMING LANGUAGE

Bjarne Steensgaard Morten Marquard

August 7, 1990

1 Introduction

Parameter splitting is a program transformation that consists of splitting a parameter to a function into two or more parameters. The object is to reduce the amount of calculations that has to be done at run-time and in some cases to improve the readability of the program. The idea is to replace a parameter whose value is a composite datastructure with a number of variables whose values are the components of the datastructure.

Parameter Splitting was first suggested in [Sestoft 86]. The background was the Copenhagen partial evaluator `Mix`. Given an interpreter, `int`, it could compile programs, `s`, by specializing them with respect to the interpreter, i.e.

```
target := L Mix <int,s>
```

However, the residual program, `target`, had a rather unnatural way of representing values of the program, `s`.

The interpreter, `int`, is supposed to handle any program written in the programming language implemented by the interpreter. When implementing the interpreter the number of variables in the source program, `s`, is usually not known in advance and therefore the values of the variables from the program, `s`, is usually kept in a list.

The specializer `Mix` does not specialize list accesses and therefore the variables from the original program is represented in the residual program as a list of values, corresponding to the interpreters internal representation. A reasonable residual program would keep each value in a separate variable.

An additional phase to the Copenhagen specializer was suggested in [Romanenko 88]. This phase should do parameter splitting. In [Romanenko 90] a complete algorithm for a parameter splitter is presented. The language of the parameter splitter is the same as used in the Copenhagen MIX-project; a pure subset of first order LISP.

Parameter splitting is referred to as *arity raising* in the papers by Romanenko. In [Mogensen 89] the more general term *retyping* is used.

Parameter splitters have been implemented as post phases to different partial evaluators. A parameter splitter implemented by T.Mogensen is based on [Romanenko 88, Mogensen 89]. Romanenko also reports that parameter splitters have been implemented [Romanenko 90].

1.1 Parameter Splitting in a higher order language

Until recently, all *autoprojectors* (i.e. selfapplicable partial evaluators), could only handle first order functional languages. *Lambda-mix* [Gomard 89,Gomard 90,Jones 90] and *Similix-2* [Bondorf 90a] are two of the first autoprojectors for higher order languages. Similix-2 is based on the first order autoprojector *Similix* [Bondorf 90b].

In this paper we describe an implementation of a parameter splitter as a post process to Similix-2.

The algorithms described in [Romanenko 90] has been used as a basis for the work. The type and parameter access analysis has been extended to handle the higher order language. Given an application, $(exp_0\ exp_1 \ldots exp_n)$, we must first find all lambda abstractions that exp_0 might evaluate to, and then perform the analysis. Given this set of lambda abstractions, the algorithms can be generalized to the higher order case.

1.2 Outline

The rest of the paper is organized as follow. Sections 2 gives some background and introduces the main issues of this paper. In section 3 we describe what we mean by *parameter splitting*. The terms defined in this section will be used in the rest of this paper. Section 4 describes the type analysis, and in section 5 the type analysis is augmented with a closure analysis, that can find the lambda abstractions that a given expression exp_0 might evaluate to. In section 6 we describe the parameter access analysis, which is a backward analysis. In section 7 we describe how we can use the information we have obtained to actually split the parameters of functions. In section 8 we describe that lambda lifting is a possible part of parameter splitting, and in section 9 we summarize the paper.

The notation and way of presentation used in this paper in inspired by the notation used in [Romanenko 90].

1.3 Prerequisites

Some knowledge about partial evaluation is required, e.g. as presented in [Jones 85] or [Jones 89], and preferably also about higher order autoprojectors [Bondorf 90a] or [Jones 90].

2 Background and Issues

2.1 The Language

Similix-2 processes a higher order language and is an extension of the language treated by Similix [Bondorf 90b]. *Lambda abstractions* and *higher order applications* have been added to the language. The language is a subset of Scheme and can be executed directly in a Scheme environment. Similix-2 provides the user with the possibility of defining which operators to use in the language.

In the following we will consider programs written in the Similix-2 subset of the Scheme language. The exact contents of the subset will be specified by the contents of the loaded adt-files as done in Similix-2. Here is the syntax of the language:

$$
\begin{array}{lll}
pgm & ::= & ld_1\ ld_2\ \ldots\ ld_n\ fd_1\ fd_2\ \ldots\ fd_n \\
ld & ::= & \texttt{(loadt "}filename\texttt{")} \\
fd & ::= & \texttt{(define (}f\ x_1\ \ldots\ x_n\texttt{)}\ exp\texttt{)} \\
exp & ::= & x \qquad\qquad \texttt{- variable} \\
 & | & constant \\
 & | & \texttt{(if }exp_1\ exp_2\ exp_3\texttt{)} \qquad\qquad \texttt{- conditional} \\
 & | & \texttt{(}predicate\ exp_1\ \ldots\ exp_n\texttt{)} \qquad\qquad \texttt{- predicate} \\
 & | & \texttt{(}operator\ exp_1\ \ldots\ exp_n\texttt{)} \\
 & | & \texttt{(}constructor\ exp_1\ \ldots\ exp_n\texttt{)} \\
 & | & \texttt{(}selector\ exp\texttt{)} \\
 & | & \texttt{(let ((}name_1\ exp_1\texttt{)}\ \ldots\ \texttt{(}name_n\ exp_n\texttt{))}\ exp_0\texttt{)} \quad \texttt{- let-expression} \\
 & | & \texttt{(lambda (}name_1\ \ldots\ name_n\texttt{)}\ exp\texttt{)} \qquad\qquad \texttt{- abstraction} \\
 & | & \texttt{(}exp_0\ exp_1\ \ldots\ exp_n\texttt{)} \qquad\qquad \texttt{- application}
\end{array}
$$

All functions in the language are strict. The only exceptions is the `if`-statement.

In addition to the datatype "S-expressions", it is often practical to have integers, strings and possibly also vectors in the language. Exactly what datatypes, predicates, operators, selectors and constructors that are to be included in the language is specified in `adt`-files loaded by the program. The `adt`-files are loaded by the `loadt` commands.

We call all loaded functions that are not predicates, selectors or constructors for operators.

It is possible to specify constant values of loaded "builtin" (basic) datatypes (e.g. number or string). Further it is possible to specify constant values of loaded abstract datastructures (e.g. triple or matrix). All constants must be first order constants.

2.2 Selector and constructors

As mentioned in [Bondorf 90b, page 33] a consequence of the open-ended design of user defined operators is that parameter splitting is no longer restricted to handling the operators `cons`, `car` and `cdr`, but may handle any (user) defined abstract data types that can be split.

Not all possible user defined sorts can be split. We have to define rules about the constructors and selectors of the sorts that are to be split. As for the binary constructor `cons` we have the selectors `car` and `cdr` selecting the first and the second component of the `cons`-element. We can write this rule as `cons(car cdr)`. We require that a corresponding rule must be defined for constructors and selectors in the same sort, if the sort is intended to be split. We also assume that there is only one constructor associated with each sort.

2.3 Mono- or polyvariant splitting

At some point we have to decide whether we will do parameter splitting in a mono- or polyvariant way. For example, given 3 functions, `f`, `g` and `h`, if `f` and `g` can both be applied to some parameters at one application point, and `g` and `h` can both be applied at another application point, in the monovariant case all 3 functions must be split in the same way. In the polyvariant case, we can duplicate the function `g` and split one copy of `g` in the same way as `f` and the other copy of `g` in the same way as `h`. Since the language is higher order this is not as simple as it sounds.

We will only consider splitting functions in a monovariant way.

3 Splitting a parameter to a function

By *parameter splitting* we mean replacing a parameter whose value is a composite data-structure with a number of variables whose values are the components of the datastructure. The following illustrates the principle.

Suppose the definition of a function f in a program has the form

$$\texttt{(define (}f\ \ldots\ x_k\ \ldots\texttt{)}\ exp\texttt{)}.$$

and we have constructors (e.g. `cons`) and selectors (e.g. `car` and `cdr`) for administrating data-structures. The phrase "splitting of the function f's k'th parameter with respect to the constructor c" will be used to refer to a two-step transformation of the definition and all applications of f.

At the first step, the original definition

$$\texttt{(define (}f\ \ldots\ x_k\ \ldots\texttt{)}\ exp\texttt{)}$$

is replaced by

$$\texttt{(define (}f\ \ldots\ x_k^{(1)}\ x_k^{(2)}\ \ldots\ x_k^{(n)}\ldots\texttt{)}\ exp[x_k \rightarrow (c\ x_k^{(1)}\ x_k^{(2)}\ \ldots\ x_k^{(n)})]\texttt{)}$$

where "$exp[x_k \rightarrow (c\ x_k^{(1)}\ x_k^{(2)}\ \ldots\ x_k^{(n)})]$" denotes the expression obtained from exp by replacing x_k with $(c\ x_k^{(1)}\ x_k^{(2)}\ \ldots\ x_k^{(n)})$. The new formal parameters, $x_k^{(1)}\ x_k^{(2)}\ \ldots\ x_k^{(n)}$, must all be different from all global variables and all the other formal parameters to the function f. The constructor c is assumed to be n-ary. The transformation might make certain local optimizations possible.

At the second step, all calls of the function f:

$$(f\ \ldots\ exp_k\ \ldots)$$

are replaced by

$$(f\ \ldots\ (sel_1\ exp_k)\ (sel_2\ exp_k)\ \ldots\ (sel_n\ exp_k)\ \ldots).$$

The selectors sel_1, sel_2, ..., sel_n all select a part of the data-structure produced by the constructor c.

It is possible, if need be, to reconstruct the original value of the parameter x_k. This can be done by evaluating an expression of the form $(c\ x_k^{(1)}\ x_k(2)\ \ldots\ x_k^{(n)})$.

Of course the new formal parameters may be split again. This may continue to arbitrary depth. In practice we will not first split a parameter with respect to one constructor and then split one of the new formal parameters with respect to another constructor, but rather split the whole thing at once.

It is assumed that the selectors sel_1, sel_2, ..., sel_n all select a part of a data-structure that exactly matches one of the parameters to the constructor of the data-structure and that there are as many selectors as parameters to the constructor. We will in fact only consider splitting variables with respect to constructors satisfying the following condition. The constructor must belong to a sort having a set of selectors where each selector selects what corresponds to one of the parameters to the constructor. In other words, the constructor and selectors in the sort must satisfy this equation:

$$\begin{aligned}(c\ exp_1\ \ldots\ exp_n) = (c\ &(sel_1\ (c\ exp_1\ \ldots\ exp_n))\\ &(sel_2\ (c\ exp_1\ \ldots\ exp_n))\\ &\vdots\\ &(sel_n\ (c\ exp_1\ \ldots\ exp_n)))\end{aligned}$$

Functions in the adt-files belonging to sorts not satisfying this condition are classified as predicates or operators.

Example 1 Splitting a formal parameter
We consider the program

```
(loadt "scheme.adt")
(define (f x) (g (cons x x)))
(define (g u) (cdr u))
```

The standard Scheme constructor cons and constructors car and cdr are defined in the file "scheme.adt" as well as other standard functions. We will split the formal parameter u of the function g with respect to the constructor cons. After performing the transformations, we have the program

```
(loadt "scheme.adt")
(define (f x) (g (car (cons x x)) (cdr (cons x x))))
(define (g u1 u2) (cdr (cons u1 u2)))
```

This program can evidently be optimized. After local optimization we get:

```
(loadt "scheme.adt")
(define (f x) (g x x))
(define (g u1 u2) u2)
```

The splitting of the parameter u has left us with an unused parameter to the function g. Further analysis may detect this and the parameter can be removed.

Splitting should only be performed if it can be guaranteed that no error is introduced into the program. That implies that we should not split a parameter that might be assigned atomic values.

Example 2 Erroneously splitting an atomic value
It would not be correct to split the parameter to the function g with respect to the constructor cons in this program:

```
(loadt "scheme.adt")
(define (f x) (g 'a))
(define (g u) u)
```

If the splitting were performed, the program would look like this:

```
(loadt "scheme.adt")
(define (f x) (g (car 'a) (cdr 'a)))
(define (g u1 u2) (cons u1 u2))
```

It it obvious that the two programs are not equivalent. The original program would have given the atom `'a` as a result while the transformed program will result in an error-condition, trying to take car or cdr of the atom `'a`.

To ensure that it is possible to split a parameter without introducing errors in the program, we need a description of the structure of the actual parameters to functions. We will call this a description of the parameter types.

4 Analysis of the parameter types

To describe the structure of values that can be used as parameters, we use the following set of types:

$$
\begin{aligned}
t &::= \bot \\
&\quad | \ u \\
u &::= any \\
&\quad | \ atom(\mathcal{A}) \\
&\quad | \ lambda(\mathcal{L}) \\
&\quad | \ sortdesc(u_1, \ u_2, \ \ldots)
\end{aligned}
$$

The symbol $\mathcal{A}$ denotes an atomic value in the used subset of Scheme, the symbol $\mathcal{L}$ denotes a lambda abstraction, and *sortdesc* denotes a sort-descriptor, where a sort is an (abstract) data-structure in the used subset of Scheme. Different atomic values are denoted by different $\mathcal{A}$'s and different lambda abstractions are denoted by different $\mathcal{L}$'s.

We use the set of types to describe what we want to know about the structure of values. If we don't know or don't want to know anything about the structure of a value we use the type *any* to describe the structure.

In Scheme, there is a sort consisting of pairs. It has a constructor cons and selectors car and cdr. Besides constructors and selectors there might be predicates (e.g. pair?) and operators associated with a sort. In this paper we will use the name *pair* as the name of the sort of pairs.

A type describing the structure of the value obtained by evaluating the expression (cons nil nil) is $pair(atom(nil), \ atom(nil))$. The structure of the value may also be described by the types $pair(any, \ any)$ or any.

We equip the set of types with a partial ordering " $\sqsubseteq$ " recursively defined by the following rules:

- $t \sqsubseteq any$, for all types t.
- $\perp \sqsubseteq t$, for all types t.
- $sortdesc(t'_1, t'_2, \ldots) \sqsubseteq sortdesc(t''_1, t''_2, \ldots)$, if $t'_1 \sqsubseteq t''_1, t'_2 \sqsubseteq t''_2, \ldots$

If $t' \sqsubseteq t''$ and $t' \neq t''$, the type t'' is said to be *more general* than the type t'.

It is easily seen that there is no chain of infinite height in the set of types.

4.1 Combining type information

We want to use the type information to determine how every variable can be split. Proper use of the types should prevent erroneous splitting of atomic values.

If an analysis reveal that a particular formal parameter can be assigned values that have either the structure

```
pair(atom(nil),
     pair(atom(nil),
          atom(nil)))
```

or the structure

```
pair(pair(atom(nil),
          atom(nil)),
     atom(nil))
```

then splitting is only possible with respect to the outermost pair. An attempt to split one of the inner pairs after performing the first splitting, will result in attempting to split an atomic value.

If the analysis reveals that a particular variable can be assigned values that have either the structure (where "_" represents an arbitrary type)

```
pair(_, _)
```

or the structure

```
triple(_, _, _)
```

then the splitting would have to depend on the actual implementation of the abstract datatypes *pair* and *triple*. Since splitting should be independent of such details, we will not attempt to split the variable in this situation.

If we take the least upper bound (e.g. as described in [Schmidt 86]) of the two first types we get the type:

```
pair(any, any).
```

Equivalently we get the type

any

if we take the least upper bound of the type with the structure *pair*(_, _) and the type with the structure *triple*(_, _, _).

We see that these types are good substitutes for the types they were derived from, if we only need the type information to decide how to split variables. This result can be generalized to all types. That means that we only need the least upper bound of the set of types instead of the whole set of types describing the structures of the values that can be assigned to a particular variable.

We use the symbol $\sqcup$ to denote the least upper bound operator.

Another way to go about the problem is to duplicate functions called with actual parameters with different structures. This amounts to performing polyvariant splitting. We believe that this would destroy the structure of the program completely and furthermore give very large and almost unreadable programs. We intend to take the least upper bound of types to be combined (i.e. perform monovariant splitting).

4.2 Preventing call-duplication

If the type analysis reveals that one particular variable is always assigned a value with a composite structure, then it is still not sure that splitting will be an advantage. If the actual parameter expression is a call of a function, then splitting the parameter may result in code duplication and call duplication.

Example 3 Splitting can result in call-duplication
We consider the following program:

```
(loadt "scheme.adt")
(define (f z) (swap (unzip z nil nil)))
(define (unzip u x y)
   (if (null? u)
       (cons x y)
       (unzip (cdr u)
              (cons (car (car u)) x)
              (cons (cdr (car u)) y))))
(define (swap v) (cons (cdr v) (car v)))
```

It is evident that any result produced by the function `unzip` can be described by the type *pair*(*any*, *any*). This type will also be assigned to the parameter `v` of the function `swap`. We are thus allowed to split `v`. Splitting `v` gives the program

```
(loadt "scheme.adt")
(define (f z)
   (swap (car (unzip z nil nil))
         (cdr (unzip z nil nil))))
(define (unzip u x y)
   (if (null? u)
       (cons x y)
       (unzip (cdr u)
```

```
                (cons (car (car u)) x)
                (cons (cdr (car u)) y))))
(define (swap v1 v2) (cons v2 v1))
```

We see that the transformation has resulted in two copies of the expression

```
(unzip z nil nil).
```

This is bad since it leads to repeated evaluation of the expression. One could of course introduce a let-expression, but this is not always desirable since it doesn't lead to any local optimizations.

The risk of code duplication and repeated evaluation can be avoided by ensuring that all selectors produced by parameter splitting are eliminatable by local optimization.

This can be done by assuming that all functions, operators and predicates return values of type *any*. The structure of the values assigned to variables will then only depend on explicitly written constructors and selectors thus ensuring that optimization is possible whenever a variable is split.

Equivalently we must describe the result of `let`-expressions and conditionals with the type *any*.

4.3 Generating the type-information

We assume that we have a function *Type* that given a (first order) constant returns the type of the constant. How this function is to be implemented will not be addressed.

If x is a variable in the program then we want to find the least general type describing the structure of all the values that can be assigned to x when the program is run. This can be done by *abstract interpretation* of the program, which amounts to performing the program's computations using abstract values in place of the actual ones.

Since the language is higher order there might be applications where the function to be applied might be one of several possible. We therefore need some kind of information about which functions can be applied at a given application point. A way to obtain this information will be explained in section 5. For the moment we just assume that if *exp* is an expression, then the function *closure-set*(*exp*) gives us a set containing the names of all the lambda abstractions that the expression *exp* might evaluate to.

We assume that every lambda abstraction has a unique name. The names can be internally generated labels (for "nameless" lambda expressions) or names of functions. This assumption serves only technical reasons and does not restrict the language in any way.

We have here used the term *lambda abstraction* to denote either a function specified in the `adt`-file, a user defined (named) function or a user defined (nameless) lambda abstraction. We will continue to use the term in this sense throughout the paper.

Suppose we have a program defining functions with the names $f_1, \ldots, f_h$. Let F be the set containing the names of these functions and all nameless lambda abstractions in the program, and for each $f \in F$, *param*(f,j) be its j'th parameter, *arity*(f) be its arity, and *body*(f) be its body. If f is a named function then the definition of f has the form:

$$\texttt{(define } (f\ param(f,1)\ \ldots\ param(f,arity(f)))\ body(f)\texttt{)}$$

Let *VName* be the set of all variable names and let

$$\theta \in Env = VName \rightarrow Type$$

be an *environment* with information about the type of each variable that can appear in a given function, and let

$$\alpha \in ArgDescr = F \rightarrow Env$$

be an *argument type description* describing the types assigned to each function's parameters.

We now define two functions R and A to perform the abstract interpretation.

The function R, given an expression *exp* and an environment θ computes the type of what the expression evaluates to.

$$\mathtt{R} \in exp \rightarrow Env \rightarrow type$$

$$\mathtt{R}[\![x]\!]\ \theta = \theta(x)$$

$$\mathtt{R}[\![constant]\!]\ \theta = Type[constant]$$

$$\mathtt{R}[\![\texttt{(if } exp_0\ exp_1\ exp_2\texttt{)}]\!]\ \theta = any$$

$$\mathtt{R}[\![\texttt{(}predicate\ exp_1\ \ldots\ exp_n\texttt{)}]\!]\ \theta = any$$

$$\mathtt{R}[\![\texttt{(}operator\ exp_1\ \ldots\ exp_n\texttt{)}]\!]\ \theta = any$$

$$\mathtt{R}[\![\texttt{(}constructor\ exp_1\ \ldots\ exp_n\texttt{)}]\!]\ \theta = \begin{cases} \bot & \texttt{if any one of R}[\![exp_1]\!]\ \theta\ \ldots\ \mathtt{R}[\![exp_n]\!]\ \theta \texttt{ are } \bot \\ sortdesc(\mathtt{R}[\![exp_1]\!]\ \theta,\ \ldots,\ \mathtt{R}[\![exp_n]\!]\ \theta) & \texttt{otherwise} \end{cases}$$

$$\mathtt{R}[\![\texttt{(}selector\ exp\texttt{)}]\!]\ \theta = \begin{cases} \bot & \texttt{if R}[\![exp]\!]\ \theta = \bot \\ t_i & \texttt{if } R[\![exp]\!]\ \theta = sortdesc(t_1,\ \ldots,\ t_h) \texttt{ and} \\ & \quad \texttt{sort}(selector) = \texttt{sort}(sortdesc) \texttt{ and} \\ & \quad selector \texttt{ is the } i\texttt{'th selector in this sort.} \\ any & \texttt{otherwise} \end{cases}$$

$$\mathtt{R}[\![\texttt{(let ((}name_1\ exp_1\texttt{)}\ \ldots\ \texttt{(}name_n\ exp_n\texttt{))}\ exp_0\texttt{)}]\!]\ \theta = any$$

$$\mathtt{R}[\![\texttt{(lambda (}name_1\ \ldots\ name_n\texttt{)}\ exp\texttt{)}]\!]\ \theta = lambda(label)\texttt{, where}$$

label is the name of the lambda abstraction

$$\mathtt{R}[\![\texttt{(}exp_0\ exp_1\ \ldots\ exp_n\texttt{)}]\!]\ \theta = any$$

In the equation for lambda abstractions we have used the symbol *label* to denote the name of the lambda abstraction in focus. This terminology will also be used later.

The expression, exp_0, in the last equation may of course evaluate to a constructor, a predicate or an operator. We will of course not use the last equation in these cases. This way of reasoning will also be used later.

The function A, given an expression *exp*, an environment θ, and an argument type description α, computes a new approximation to the final description of each function's

parameter values. It does this by traversing the expression and updating α as it encounters applications. It uses a global environment, θ_0, mapping names of all defined functions to the corresponding lambda abstractions.

We use the notation

$$\mathcal{M}\,[name \mapsto something]$$

to denote the mapping $\mathcal{M}$ with the addition (or modification) that *name* maps into the value *something*.

$$\mathtt{A} \in exp \to Env \to ArgDescr \to ArgDescr$$

$$\mathtt{A}\,[\![x]\!]\;\theta\;\alpha = \alpha$$
$$\mathtt{A}\,[\![constant]\!]\;\theta\;\alpha = \alpha$$
$$\mathtt{A}\,[\![(\mathtt{if}\ exp_0\ exp_1\ exp_2)]\!]\;\theta\;\alpha = \mathtt{A}\,[\![exp_0]\!]\;\theta\;\alpha \sqcup \mathtt{A}\,[\![exp_1]\!]\;\theta\;\alpha \sqcup \mathtt{A}\,[\![exp_2]\!]\;\theta\;\alpha$$
$$\mathtt{A}\,[\![(predicate\ exp_1\ \ldots\ exp_n)]\!]\;\theta\;\alpha = \textstyle\bigsqcup_{i=1}^{n} \mathtt{A}\,[\![exp_i]\!]\;\theta\;\alpha$$
$$\mathtt{A}\,[\![(operator\ exp_1\ \ldots\ exp_n)]\!]\;\theta\;\alpha = \textstyle\bigsqcup_{i=1}^{n} \mathtt{A}\,[\![exp_i]\!]\;\theta\;\alpha$$
$$\mathtt{A}\,[\![(constructor\ exp_1\ \ldots\ exp_n)]\!]\;\theta\;\alpha = \textstyle\bigsqcup_{i=1}^{n} \mathtt{A}\,[\![exp_i]\!]\;\theta\;\alpha$$
$$\mathtt{A}\,[\![(selector\ exp)]\!]\;\theta\;\alpha = \mathtt{A}\,[\![exp]\!]\;\theta\;\alpha$$
$$\mathtt{A}\,[\![(\mathtt{let}\ ((name_1\ exp_1)\ \ldots\ (name_n\ exp_n))\ exp_0)]\!]\;\theta\;\alpha =$$
$$\left\{\textstyle\bigsqcup_{i=1}^{n} \mathtt{A}\,[\![exp_i]\!]\;\theta\;\alpha\right\} \sqcup \mathtt{A}\,[\![exp_0]\!]\;\theta_{new}\;\alpha,\ \mathtt{where}$$
$$\theta_{new} = \theta[name_i \mapsto \mathtt{R}\,[\![exp_i]\!]\;\theta]_{i=1,\ldots,n}$$
$$\mathtt{A}\,[\![(\mathtt{lambda}\ (name_1\ \ldots\ name_n)\ exp)]\!]\;\theta\;\alpha = \mathtt{A}\,[\![exp]\!]\;\theta_{new}\;\alpha,\ \mathtt{where}$$
$$\theta_{new} = \theta[\alpha(label)]$$
$$\mathtt{A}\,[\![(exp_0\ exp_1\ \ldots\ exp_n)]\!]\;\theta\;\alpha =$$
$$\mathtt{A}\,[\![exp_0]\!]\;\theta\;\alpha \sqcup \alpha_{new}\,[f \mapsto (\alpha_{new}(f) \sqcup \theta_{new,f})],\ \mathtt{where}$$
$$f \in closure\text{-}set(exp_0)\ \mathtt{and}\ arity(f) = n\ \mathtt{and}$$
$$\alpha_{new} = \textstyle\bigsqcup_{j=1}^{n} \mathtt{A}\,[\![exp_j]\!]\;\theta\;\alpha\ \mathtt{and}$$
$$\theta_{new,f} = \left[param(f,j) \mapsto \mathtt{R}\,[\![exp_j]\!]\;\theta\right]_{j=1,\ldots,n}$$

We want a final argument type description, α, that is consistent and as low as possible. This description can be determined by finding the least fixed point for the following system of simultaneous equations and relations:

$$\alpha = \textstyle\bigsqcup \{\mathtt{A}\,[\![body(f)]\!]\;\alpha(f)\;\alpha\}_{f \in F}, \qquad \alpha \geq \alpha_0$$

where α_0 is defined as follows:

$$\alpha_0 = \left[f_1 \mapsto \theta_0[param(f_1,j) \mapsto any]_{j=1,\ldots,\mathtt{arity}(f_1)}\right] \sqcup$$
$$\left[f \mapsto \theta_0[param(f,j) \mapsto \bot]_{j=1,\ldots,\mathtt{arity}(f)}\right]_{f \in F}$$

The description α_0 assigns the type *any* to the parameters of the goal function, f_1, to prevent these parameters from being split. All other parameters are assigned the type $\bot$, indicating that we have no *a priori* knowledge about their structure.

The least fixed point for the system of equations above does exist and can be reached in finite time. This is because the functions R and A are monotonic and because for any given (finite) program, there does not exist any chains of infinite height in the ordered set of values that can be assigned to the α's.

5 Generating the closure-information

The method used in this section for obtaining the closure-information is inspired by the method used in [Bondorf 90a]. The method was originally introduced in [Sestoft 88]. The actual way of obtaining the information is drasticly modified to be easier to read and to understand. Furthermore the closure analysis is combined with the abstract interpretation to ease the task of performing the computations.

The closure information that we need is just another kind of type information. We need to know all the lambda abstractions that a given expression might evaluate to. In the above analysis we generalized the type information, throwing some information away. If we instead collect and keep all information about lambda abstractions assigned to variables, then we have what we need. We can use the same type system as before with the modification that we can annotate every part of the type tree with information about which lambda abstractions might be assigned to that part of the tree. The set of annotated types look like this:

$$
\begin{array}{lll}
t & ::= & \bot\{\ \} \\
 & | & u \\
u & ::= & any\{Annotations\} \\
 & | & atom(\mathcal{A})\{\ \} \\
 & | & lambda(\mathcal{L})\{\mathcal{L}\} \\
 & | & sortdesc(u_1,\ u_2,\ \ldots)\{\ \} \\
Annotations & ::= & Set\ of\ names.
\end{array}
$$

The annotations are sets of names of lambda abstractions which may appear in the corresponding part of the value described by the type.

To retrieve information about the annotations of the type tree, we use two functions *collect* and *collect-all*. The function *collect* retrieves the set of lambda abstractions that the root of the tree is annotated with. The function *collect-all* retrieves the set of all lambda abstractions that some part of the type tree is annotated with. In practice *collect* will work as *collect-all* since the tree won't have any leaves, when *collect* is applied.

When generating the closure-information we cannot afford ignoring the type of the values returned by functions, since the result might be a lambda abstraction. We therefore introduce a *result type description*:

$$\rho \in ResDescr = F \rightarrow AnnotatedType$$

assigning a type to each function's result. The definitions of θ and α also have to be changed to reflect the annotation of the types. These changes are trivial.

It is assumed that predicates defined in the adt-files return values of type *any* with no annotations. Operators defined in the adt-files are assumed to return values of type *any* annotated with all the names that are part of the annotated types describing the parameters to the operator.

We define two functions B and D serving the same purposes as the functions R and A.

The function B, given an expression *exp*, an environment θ, and a result type description ρ, computes the type of the expression's result.

$$\mathtt{B} \in exp \to Env \to ResDescr \to AnnotatedType$$

$$\mathtt{B}[\![x]\!]\ \theta\ \rho = \theta(x)$$

$$\mathtt{B}[\![constant]\!]\ \theta\ \rho = Type[constant]\{\ \}$$

$$\mathtt{B}[\![(\mathtt{if}\ exp_0\ exp_1\ exp_2)]\!]\ \theta\ \rho = any\left\{\bigcup_{i=1}^{2} collect\text{-}all(\mathtt{B}[\![exp_i]\!]\ \theta\ \rho)\right\}$$

$$\mathtt{B}[\![(predicate\ exp_1\ \ldots\ exp_n)]\!]\ \theta\ \rho = any\{\ \}$$

$$\mathtt{B}[\![(operator\ exp_1\ \ldots\ exp_n)]\!]\ \theta\ \rho = any\left\{\bigcup_{i=1}^{n} collect\text{-}all(\mathtt{B}[\![exp_i]\!]\ \theta\ \rho)\right\}$$

$$\mathtt{B}[\![(constructor\ exp_1\ \ldots\ exp_n)]\!]\ \theta\ \rho = \begin{cases} \bot\{\ \} & \mathtt{if\ any\ one\ of\ B}[\![exp_1]\!]\ \theta\ \rho\ \ldots\ \mathtt{B}[\![exp_n]\!]\ \theta\ \rho\ \mathtt{are}\ \bot\{\ \} \\ sortdesc(\mathtt{B}[\![exp_1]\!]\ \theta\ \rho,\ \ldots,\ \mathtt{B}[\![exp_n]\!]\ \theta\ \rho)\{\ \} & \mathtt{otherwise} \end{cases}$$

$$\mathtt{B}[\![(selector\ exp)]\!]\ \theta\ \rho = \begin{cases} \bot\{\ \} & \mathtt{if\ B}[\![exp]\!]\ \theta\ \rho = \bot\{\ \} \\ t_i & \mathtt{if\ B}[\![exp]\!]\ \theta\ \rho = sortdesc(t_1,\ \ldots,\ t_n)\{\ \}\ \mathtt{and} \\ & \quad \mathtt{sort}(selector) = sortdesc\ \mathtt{and} \\ & \quad selector\ \mathtt{is\ the}\ i\mathtt{'th\ selector\ to\ this\ sort.} \\ any\{collect\text{-}all(\mathtt{B}[\![exp]\!]\ \theta\ \rho)\} & \mathtt{otherwise} \end{cases}$$

$$\mathtt{B}[\![(\mathtt{let}\ ((name_1\ exp_1)\ \ldots\ (name_n\ exp_n))\ exp_0)]\!]\ \theta\ \rho = any\{collect\text{-}all(\mathtt{B}[\![exp_0]\!]\ \theta_{new}\ \rho)\},\ \mathtt{where}\ \theta_{new} = \theta\left[name_i \mapsto \mathtt{B}[\![exp_i]\!]\ \theta\ \rho\right]_{i=1,\ldots,n}$$

$$\mathtt{B}[\![(\mathtt{lambda}\ (name_1\ \ldots\ name_n)\ exp)]\!]\ \theta\ \rho = lambda(label)\{label\}$$

$$\mathtt{B}[\![(exp_0\ exp_1\ \ldots\ exp_n)]\!]\ \theta\ \rho = \bigsqcup\{\rho(f)\},\ \mathtt{where}\ f \in collect(\mathtt{B}[\![exp_0]\!]\ \theta\ \rho)\ \mathtt{and}\ arity(f) = n$$

It should be noted, that we have assumed that an operator cannot return a value containing a lambda abstraction not contained in any of the parameters to the operator.

The function D, given an expression *exp*, an environment θ, an argument type description α, and a result type description ρ, computes a new approximation to the final description of each function's parameter values.

$$\mathtt{D} \in exp \to Env \to ArgDescr \to ResDescr \to ArgDescr$$

$$\mathtt{D}[\![x]\!]\ \theta\ \alpha\ \rho = \alpha$$

$$\mathtt{D}[\![constant]\!]\ \theta\ \alpha\ \rho = \alpha$$

$$\mathtt{D}[\![(\mathtt{if}\ exp_0\ exp_1\ exp_2)]\!]\ \theta\ \alpha\ \rho = \mathtt{D}[\![exp_0]\!]\ \theta\ \alpha\ \rho \sqcup \mathtt{D}[\![exp_1]\!]\ \theta\ \alpha\ \rho \sqcup \mathtt{D}[\![exp_2]\!]\ \theta\ \alpha\ \rho$$

$$\mathtt{D}[\![(predicate\ exp_1\ \ldots\ exp_n)]\!]\ \theta\ \alpha\ \rho = \bigsqcup_{i=1}^{n} \mathtt{D}[\![exp_i]\!]\ \theta\ \alpha\ \rho$$

$$\mathtt{D}[\![(operator\ exp_1\ \ldots\ exp_n)]\!]\ \theta\ \alpha\ \rho = \bigsqcup_{i=1}^{n} \mathtt{D}[\![exp_i]\!]\ \theta\ \alpha\ \rho$$

$$\mathtt{D}[\![(constructor\ exp_1\ \ldots\ exp_n)]\!]\ \theta\ \alpha\ \rho = \bigsqcup_{i=1}^{n} \mathtt{D}[\![exp_i]\!]\ \theta\ \alpha\ \rho$$

$$\mathtt{D}[\![(selector\ exp)]\!]\ \theta\ \alpha\ \rho = \mathtt{D}[\![exp]\!]\ \theta\ \alpha\ \rho$$

$$\mathtt{D}[\![(\mathtt{let}\ ((name_1\ exp_1)\ \ldots\ (name_n\ exp_n))\ exp_0)]\!]\ \theta\ \alpha\ \rho =$$
$$\left\{\bigsqcup_{i=1}^{n} \mathtt{D}[\![exp_i]\!]\ \theta\ \alpha\ \rho\right\} \sqcup \mathtt{D}[\![exp_0]\!]\ \theta_{new}\ \alpha\ \rho,\ \mathtt{where}$$
$$\theta_{new} = \theta[name_i \mapsto \mathtt{B}[\![exp_i]\!]\ \theta\ \alpha\ \rho]_{i=1,\ldots,n}$$

$$\mathtt{D}[\![(\mathtt{lambda}\ (name_1\ \ldots\ name_n)\ exp)]\!]\ \theta\ \alpha\ \rho = \mathtt{D}[\![exp]\!]\ \theta_{new}\ \alpha\ \rho,\ \mathtt{where}$$
$$\theta_{new} = \theta[\alpha(label)]$$

$$\mathtt{D}[\![(exp_0\ exp_1\ \ldots\ exp_n)]\!]\ \theta\ \alpha\ \rho =$$
$$\mathtt{D}[\![exp_0]\!]\ \theta\ \alpha\ \rho \sqcup \alpha_{new}[f \mapsto (\alpha_{new}(f) \sqcup \theta_{new,f})],\ \mathtt{where}$$
$$f \in collect(\mathtt{B}[\![exp_0]\!]\ \theta\ \alpha\ \rho)\ \text{and}\ arity(f) = n\ \text{and}$$
$$\alpha_{new} = \bigsqcup \left\{\mathtt{D}[\![exp_j]\!]\ \theta\ \alpha\ \rho\right\}_{j=1,\ldots,n}\ \text{and}$$
$$\theta_{new,f} = \theta_0 \left[param(f,j) \mapsto \mathtt{B}[\![exp_j]\!]\ \theta\right]_{j=1,\ldots,n}$$

We want a final argument type description, α, that is consistent and as low as possible. This description can be determined by finding the least fixed point for the following system of simultaneous equations and relations:

$$\alpha = \bigsqcup \{\mathtt{D}[\![body(f)]\!]\ \alpha(f)\ \alpha\}_{f \in F}, \qquad \alpha \geq \alpha_0$$
$$\rho = [f \mapsto any\{collect\text{-}all(\mathtt{B}[\![body(f)]\!]\ \alpha(f)\ \rho\ \}]_{f \in F}, \qquad \rho \geq \rho_0$$

where α_0 and ρ_0 are defined as follows:

$$\alpha_0 = \begin{bmatrix} f_1 \mapsto [param(f_1,j) \mapsto any\{\ \}]_{j=1,\ldots,arity(f_1)} \end{bmatrix} \sqcup$$
$$\begin{bmatrix} f \mapsto [param(f,j) \mapsto \bot\{\ \}]_{j=1,\ldots,arity(f)} \end{bmatrix}_{f \in F}$$
$$\rho_0 = [f \mapsto \bot\{\ \}]_{f \in F}$$

The description α_0 assigns the type *any* to the parameters of the goal function, f_1, to prevent these parameters from being split. All other parameters are assigned the type $\bot$, indicating that we have no *a priori* knowledge about their structure. The description ρ_0 assigns the type $\bot$ to the results of all functions.

The result type description ρ is defined to reflect the assumption that all functions return values of the type *any*. This is done to prevent call-duplication when splitting, as described in section 4.2. The result type is (of course) annotated with all the lambda abstractions that might appear in some part of the value returned by the function.

Special care must be taken when deciding how to split functions that are used as parameters to other functions. If the closure analysis reveals that at a particular application point two different functions might be used then the parameters of these two functions *must* be split in the same way. If they are not, an error may be introduced into the program.

Example 4 Splitting higher order functions
Consider the following program:

```
(loadt "scheme.adt")
(define (f x)
   (cons (g h (cons x x))
         (cons (g j (cons x x))
               (h x))))
(define (g x y) (x y))
(define (h u) (car u))
(define (j v) (cdr v))
```

The type analysis will assign the type (leaving out the annotations) *pair*(*any*, *any*) to the formal parameter y, the type *any* to the formal parameter u and the type *pair*(*any*, *any*) to the formal parameter v. If we attempt to split the parameters y and v with respect to the constructor cons we get the program:

```
(loadt "scheme.adt")
(define (f x)
   (cons (g h x x)
         .(cons (g j x x)
                (h x))))
(define (g x y1 y2) (x (cons y1 y2)))
(define (h u) (car u))
(define (j v1 v2) (cdr (cons v1 v2)))
```

This program is clearly erroneous. j will be applied with the wrong arity in the body of g. We thus see that if more than one function might be used at a particular application point, then all parameters to these functions must be split the same way.

We can get around this problem by assigning the same types to the formal parameters of the functions that might occur at the same application point. The type to be used instead of the "original" types can be the least upper bound of the original types or any type that is more general than the original types.

Determining which functions may occur at an application point can be done in several ways. One way is to traverse the type tree and put all lambda abstractions that a part of the tree is annotated with in a group together. If the splitting is to be monovariant then we further have to combine groups of lambda abstractions that have common elements. If the splitting can be polyvariant we can just make a copy of each lambda abstraction for each occurrence in a group and rename it. As a result, no lambda abstraction will occur in more than one group, no matter what method is used.

If we want to be sure that every introduced selector is eliminatable by local optimization, then the job of combining groups of lambda abstractions has to be done during the type analysis described above. In the monovariant case this can be done by changing the application rule for the function D to the following:

$$\mathtt{D}[\![(exp_0\ exp_1\ \ldots\ exp_n)]\!]\ \theta\ \alpha\ \rho =$$
$$\{\bigsqcup\{\rho(f')\}\} \bigsqcup \mathtt{D}[\![exp_0]\!]\ \theta\ \alpha\ \rho \bigsqcup \alpha_{new}[f \mapsto (\alpha_{new}(f) \bigsqcup \theta_{new,f})],\ \texttt{where}$$
$$f \in collect(\mathtt{B}[\![exp_0]\!]\ \theta\ \alpha\ \rho)\ \texttt{and}\ arity(f) = n\ \texttt{and}$$

$$\alpha_{new} = \bigsqcup \left\{ \mathtt{D}\,[\![exp_j]\!]\ \theta\ \alpha\ \rho \right\}_{j=1,\ldots,n} \text{ and}$$
$$\theta_{new,f} = \theta_0 \left[param(f,j) \mapsto \mathtt{B}\,[\![exp_j]\!]\ \theta \right]_{j=1,\ldots,n}, \text{ and}$$
f' is a closure-super-set and $f \in f'$

By a closure-super-set we mean a minimal set of closures that has the property that there does not exist any closure not in the set that can appear at the same application point as one of the closures in the set.

If we want to ensure that functions that might be given as (part of) the result of the goal-function will never be split, then we have to add a rule that ensures that all parameters of all functions in the set:

$$collect(\rho(f_1))$$

are described by the type *any*.

6 The usefulness of parameter splitting

One purpose of parameter splitting is to pave the way to local optimization. Parameter splitting based only on the type analysis described above, can cause "arity overraising", by increasing the number of parameters without reducing the number of selectors in the program.

Example 5 Arity overraising
Consider the following program:

```
(loadt "scheme.adt")
(define (f x) (rev x (cons 'a nil)))
(define (rev u v)
        (if (equal?  u nil)
            v
            (rev (cdr u) (cons (car u) v))))
```

The type analysis tells us that the parameter x of the function `f` has the type *any*, and that the parameters u and v of the function `rev` has the types *any* and *pair(any, any)*. It is thus possible to split the second parameter, v, to the function `rev`. After performing the transformation rev[v → (cons v1 v2)] we get the program:

```
(loadt "scheme.adt")
(define (f x) (rev x 'a nil))
(define (rev u v1 v2)
        (if (equal?  u nil)
            (cons v1 v2)
            (rev (cdr u) (car u) (cons v1 v2))))
```

We see from the program obtained from splitting v, that no selectors have disappeared, i.e. it was of no use to perform the splitting.

We have used the notation "rev[v → (cons v1 v2)]" to denote the splitting of the parameter v of the function `rev` with respect to the constructor `cons`.

The example shows us that if there is no attempt to address the subparts of a given parameter then it is to no use splitting the parameter. We therefore need an analysis revealing what parts of a given parameter are addressed. We can then use this information for deciding when not to split a given parameter.

It is simple to avoid splitting some part of a parameter. Let the parameter x be described by the type t. If some part of t has the form $sortdesc(t_1, \ldots, t_m)$ and we don't want to split this part, this can be avoided by replacing $sortdesc(t_1, \ldots, t_m)$ with *any*.

If we lose all information about the structure of part of a value (i.e. it's described by the type *any*) then we can't tell if it's possible to split it and consequently we will not do so.

In the above example, the splitting rev[v → (cons v1 v2)] can be avoided by generalizing the type of the variable v to *any*.

We now face the problem of when to retain or generalize a *sortdesc* in the type of a parameter, i.e. when to perform the splitting corresponding the *sortdesc* and when not to. The weakest reasonable requirement is that a *sortdesc* in the type description of the parameters should be retained (only) if it causes a selector in the program to disappear. This is to ensure that local optimization is possible. The strongest requirement is that a *sortdesc* in the type description of the parameters should be retained only if it does not lead to introduction of new constructors in the body of the lambda abstraction.

Example 6 Generalizing types
Consider the example:

```
(loadt "scheme.adt")
(define (f x) (g (cons 'a (cons 'b (cons 'c 'd)))))
(define (g u)
   (cons (cdr u)
         (car (cdr u))))
```

If we split the parameter to the function g into four parameters, we will overraise the arity. If we generalize the types according to the weakest requirement listed above and then perform the splitting then we get the program:

```
(loadt "scheme.adt")
(define (f x) (g 'a 'b (cons 'c 'd)))
(define (g u1 u2 u3)
   (cons (cons u2 u3) u2))
```

If we instead generalize the types according to the strongest requirement listed above and then perform the splitting then we get the program:

```
(loadt "scheme.adt")
(define (f x) (g 'a (cons 'b (cons 'c 'd))))
(define (g u1 u2)
   (cons (car u2) u2))
```

There are obviously advantages and disadvantages connected with both solutions. The weakest requirement may result in the fastests programs in most cases but it can also lead to slower programs in special cases. The strongest requirement will never lead

to introduction of extra constructors and will therefore not have this weakness. Instead it might be too conservative in most cases. The problem seems to be analogous to the *call unfolding problem* of partial evaluation.

In [Romanenko 90] the weakest requirement is used. Inspired by this we will also use this requirement.

Given a parameter to a lambda abstraction, we now want to analyse how the parameter is accessed by selectors in the body of the lambda abstraction.

6.1 Access paths and contexts

We start with a simple case. For example, if the variable x is used in an expression:

$$(sel1\ (sel2\ (sel3\ x)))$$

then x is accessed by applying selectors in the order: $sel3$, $sel2$, $sel1$. The component of x to be accessed can be unambiguously identified by a sequence of selectors.

We define an *access path* to be a finite list (which may be empty) of selector names. The set of all access paths will be denoted by *Path*.

We use the following notation. A finite list of elements $a_1 \ldots a_n$ is written as $[a_1 \ldots a_n]$ and an empty list is written as [ ]. The concatenation of two lists $A = [a_1 \ldots a_n]$ and $B = [b_1 \ldots b_m]$ equal to $[a_1 \ldots a_n\ b_1 \ldots b_m]$ is written as $A \hat{} B$.

The formal parameter x to a function `f` can be accessed at many different places in the body of the function, i.e. by many different access paths. Therefore, instead of a single access path, we define an *access context*.

An access context, Π, is a set of access paths, which satisfies the following requirements:

1. $[\ ] \in \Pi$
2. If $\pi \hat{} [selector] \in \Pi$ then $\pi \in \Pi$

Requirement (1.) means that an attempt at accessing the parameter x as a whole must be included into the context. This requirement is useful for technical reasons. Requirement (2.) means that a subcomponent of x can only be accessed by accessing a component in which the subcomponent is included. That is, an access context is a non-empty prefix-closed set of access paths. The set of all contexts is denoted by *Context*.

6.2 Using contexts for type generalization

Let a parameter have the type t and the context Π. Then a function *GenType* can be easily defined which generalizes t in accordance with Π by replacing all $sortdesc(t_1, \ldots, t_n)$ unaccessed by Π with *any*.

$$GenType \in Type \rightarrow Context \rightarrow Type$$

$$GenType\ [t]\ \Pi = \sqcap \{GenType'\ [t]\ \pi \mid \pi \in \Pi\}$$

The $\sqcap$ is used to denote the *greatest lower bound* of a set of values (e.g. as described in [Schmidt 86]).

$$GenType' \in Type \rightarrow Path \rightarrow Type$$

$$\begin{aligned}
&GenType'\,[any]\ \pi = any\\
&GenType'\,[atom(\mathcal{A})]\ \pi = atom(\mathcal{A})\\
&GenType'\,[lambda(\mathcal{L})]\ \pi = lambda(\mathcal{L})\\
&GenType'\,[sortdesc(t_1, \ldots, t_n)]\ [\,]\ = any\\
&GenType'\,[sortdesc(t_1, \ldots, t_n)]\ ([selector]^\frown\pi)\ =\\
&\quad \begin{cases}
sortdesc(any, \ldots, any, GenType'\,[t_i]\ \pi, any, \ldots, any) & \\
\qquad \texttt{if } \mathrm{sort}(selector) = sortdesc \texttt{ and} & \\
\qquad selector \text{ is the } i\text{'th selector in the sort} & \\
any \qquad \texttt{otherwise.} &
\end{cases}\\
&GenType'\,[\bot]\ \pi = \bot
\end{aligned}$$

It should be noted that for all $t \in Type$ and all paths π the relation $t \sqsubseteq GenType'\,[t]\ \pi$ holds, therefore the set $\{GenType'\,[t]\ \pi \mid \pi \in \Pi\}$ is finite. Consequently, the *greatest lower bound* of this set does exist.

6.3 Finding the context of a parameter

Given the program with all its lambda abstraction, let us now define a function $\dot{\mathtt{C}}$ that given a parameter, an expression, and a context for the expression, finds a context for the parameter, i.e. can find the sequences of selectors accessing the parameter.

$$\mathtt{C} \in VName \rightarrow Exp \rightarrow Context \rightarrow Context$$

In order to find the contexts of all formal parameters in the program, we apply this function, C, to all parameters of each lambda abstraction with the body of the lambda abstraction as the second parameter. The contexts of these expressions are the empty context:

$$cont(f,k) = \mathtt{C}\ x\ [\![body(f)]\!]\ \{[\,]\}$$

We have here used the notation $cont(f,k)$ to denote the context of the k'th parameter of the function f.

We have to do the calculations for all parameters of both the named functions and the unnamed lambda abstractions in the program.

Given a parameter, x, let us now, step by step, analyse the different forms of expressions as given in section 2.1.

Expression ::= Variable

We have 2 possibilities: (1) the variable is x, or (2) it is not. In the first case the new context must be equal to the given context, in the second case the context must be empty.

$$\mathtt{C}\ x\ [\![y]\!]\ \Pi = \begin{cases} \Pi & \text{if } x = y \\ \{[\ ]\} & \text{otherwise} \end{cases}$$

Expression ::= Constant
The parameter x doesn't occur in the expression. The context of the constant does therefore not propagate to x. This gives us:

$$\mathtt{C}\ x\ [\![constant]\!]\ \Pi = \{[\ ]\}$$

Expression ::= (if exp_0 exp_1 exp_2)
There is more than one way to treat this expression, so let us consider an example.

Example 7 Conditional
Suppose the context, Π, equals $\{[\,], [car]\}$. It is obvious that this context does not access the conditional expression, exp_0, so the context for this must be empty. But what is the context of exp_1 and exp_2? The context, Π, will operate on the result of these expressions. Therefore we could use the context for these expressions. If we propagate the given context, it would correspond to the following rewriting of the function (suggested in [Mogensen 89]), causing code duplication but not call duplication.

$$\mathtt{(car\ (if}\ exp_0\ exp_1\ exp_2\mathtt{))} = \mathtt{(if}\ exp_0\ \mathtt{(car}\ exp_1\mathtt{)\ (car}\ exp_2\mathtt{))}$$

The transformation in the example is however rather exotic. We do not intend to include this kind of transformation. That means that we can not use the context, Π, we have found already. The context of the parameter depends only on the three subparts of the conditional expression. Therefore we end up with:

$$\mathtt{C}\ x\ [\![\mathtt{(if}\ exp_0\ exp_1\ exp_2\mathtt{)}]\!]\ \Pi = \bigcup_{i=0}^{2} \mathtt{C}\ x\ [\![exp_i]\!]\ \{[\ ]\}$$

Expression ::= (predicate exp_1 ... exp_n)
Expression ::= (operator exp_1 ... exp_n)
The given context does not access any of the expressions $exp_1 \ldots exp_n$ but access the result of performing the operation. Therefore we find the new context as the union of all the contexts of all occurrences of x in all the expressions, i.e.

$$\mathtt{C}\ x\ [\![\mathtt{(}predicate\ exp_1\ \ldots exp_n\mathtt{)}]\!]\ \Pi = \bigcup_{i=1}^{n} \mathtt{C}\ x\ [\![exp_i]\!]\ \{[\ ]\}$$

$$\mathtt{C}\ x\ [\![\mathtt{(}operator\ exp_1\ \ldots exp_n\mathtt{)}]\!]\ \Pi = \bigcup_{i=1}^{n} \mathtt{C}\ x\ [\![exp_i]\!]\ \{[\ ]\}$$

Expression ::= (constructor $exp_1 \ldots exp_n$)
There are two ways to treat constructors, depending on whether we want to perform some reductions on constructors or not.

Example 8 Reductions on constructors
Suppose we have the expression

(cdr (car (cons exp_1 exp_2)))

The expression can be reduced to the expression

(cdr exp_1)

We thus see that it is reasonable to take the context of exp_1 to be {[ ],[cdr]}. The reduction can however lead to a change in the programs termination properties. Similix-2 guarantees that it doesn't change the programs termination properties so neither will we. Therefore we won't perform the reduction.

Consequently, given a constructor, we find the new context by

$$\mathtt{C}\ x\ [\![(constructor\ exp_1\ \ldots exp_n)]\!]\ \Pi = \bigcup_{i=1}^{n} \mathtt{C}\ x\ [\![exp_i]\!]\ \{[\ ]\}$$

Expression ::= (selector exp)
Given a selector in an expression, we have to add the selector to all access paths in the given context:

$$\mathtt{C}\ x\ [\![(sel\ exp)]\!]\ \Pi = \mathtt{C}\ x\ [\![exp]\!]\ \left(\{[sel] \frown \pi \mid \pi \in \Pi\} \cup \{[\ ]\}\right)$$

Expression ::= (lambda ($v_1 \ldots v_n$) exp)
Given a lambda abstraction, how do we find the new context? This, turns out to be simple. We have 2 possibilities, (1) x is a parameter to the lambda abstraction, or (2) it does not. In the first case, the context must be empty, because the parameter can't be used in the lambda abstraction because it is hidden by a new parameter. In the second case, the context is the context in the body of the lambda abstraction:

$$\mathtt{C}\ x\ [\![(\mathtt{lambda}\ (v_1\ \ldots v_n)\ exp)]\!]\ \Pi = \begin{cases} \{[\ ]\} & \mathtt{if}\ x \in \{v_1 \ldots v_n\} \\ \mathtt{C}\ x\ [\![exp]\!]\ \{[\ ]\} & \mathtt{otherwise}. \end{cases}$$

It is possible to transform lambda-expressions in a manner similar to the exotic transformation of if-expressions described above. We will not do this.

Expression ::= (let ((v_1 exp_1) ... (v_n exp_n)) exp_0)
Given a let-expression, let us rewrite the expression:

(let ((v_1 exp_1) ... (v_n exp_n)) exp_0) =
((lambda (v_1 ... v_n) exp_0) exp_1 ... exp_n)

Given how we find the context of an application, we can find the context of a let-expression by the rewriting suggested above. The actual rule for let-expressions have to wait until we have defined a rule for applications.

Expression ::= (exp_0 exp_1 ... exp_n)
We could define the rule similar to operators, i.e.

$$\mathtt{C}\ x\ [\![(exp_0\ exp_1\ \ldots exp_n)]\!]\ \Pi = \bigcup_{i=0}^{n} \mathtt{C}\ x\ [\![exp_i]\!]\ \{[\ ]\} \qquad \textbf{WRONG!}$$

because the given context doesn't access any of the subexpressions $exp_0 \ldots exp_n$ in the application. This might lead to the conclusion, that to find the context of a given parameter, we only have to analyse the body of the function, where the parameter is defined. However, this conclusion turns out to be wrong, and we have to change the definition of C for applications. In the next subsection we'll describe why and how the definition can be changed.

6.4 Latent Selectors

Example 9 Introducing latent selectors
Consider the following simple first order program:

```
(loadt "scheme.adt")
(define (f x) (g (cons x 'a)))
(define (g u) (h u))
(define (h v) (cdr v))
```

By the rule for application suggested above, we find the following context for the parameters:

$$cont(f,1) = \{[\ ]\} \quad cont(g,1) = \{[\ ]\} \quad cont(h,1) = \{[\ ],[cdr]\}$$

We get the impression that it is useful to split $param$(h,v), since the splitting $h[v \rightarrow (cons\ v1\ v2)]$ causes a selector to disappear:

```
(loadt "scheme.adt")
(define (f x) (g (cons x 'a)))
(define (g u) (h (car u) (cdr u)))
(define (h v1 v2) v2)
```

This result is far from being satisfactory since two new selectors, `car` and `cdr`, have appeared.

The example makes us draw the conclusion that the *parameter access analysis* has to take into account not only the selectors explicitly appearing in the program, but also the *latent selectors* to be introduced by parameter splitting.

Example 10 Improving the treatment of latent selectors
Performing the parameter access analysis on the new program given in the example above, gives the result:

$$cont(f,1) = \{[\ ]\} \quad cont(g,1) = \{[\ ],[car],[cdr]\} \quad cont(h,1) = \{[\ ]\} \quad cont(h,2) = \{[\ ]\}$$

These new contexts causes us to get a better program as result:

```
(loadt "scheme.adt")
(define (f x) (g x 'a))
(define (g u1 u2) (h u1 u2))
(define (h v1 v2) v2)
```

The problem with the latent selectors has now disappeared.

We can draw the conclusion from example 10 that it would be incorrect to assume the context of each expression $exp_1 \ldots exp_n$ in an application $(exp_0\ exp_1 \ldots exp_n)$ to be $\{[\ ]\}$. If we for example have the function

```
(define (g v) (...))
```

and an application of g

```
(g exp)
```

the splitting g [v → (cons v1 v2)] would give the application

```
(g (car exp) (cdr exp))
```

i.e. the context must be changed to take this into account. All attempts at accessing `exp` due to a possible splitting of the formal parameter corresponding to `exp` in the application, must be taken into account.

Let $type(f,k)$ be the type of the k'th parameter of the function `f`. The type provides us with information about how the parameter may be split, and the context provides us with information about whether it is advantageous to perform the splitting.

Generalizing the type, $type(f,k)$, with respect to the context, $cont(f,k)$, gives a new type, $type'(f,k) = GenType\ [type(f,k)]\ cont(f,k)$, with all information about how the parameter should be split. Given the generalized type, we can define a function calculating a context which accesses all the components of the type. This is the context due to the splitting of the k'th formal parameter according to the type.

$$TypeToContext \in Type \rightarrow Context$$

$$\begin{aligned}
&TypeToContext\ [any] = \{[\ ]\}\\
&TypeToContext\ [atom(\mathcal{A})] = \{[\ ]\}\\
&TypeToContext\ [lambda(\mathcal{L})] = \{[\ ]\}\\
&TypeToContext\ [sortdesc(t_1,\ \ldots,\ t_n)] =\\
&\qquad \{[\ ]\} \cup \bigcup_{i=1}^{n} \{[sel_i]^\frown \pi \mid \pi \in TypeToContext\ [t_i]\},\ \texttt{where}\\
&\qquad \mathrm{sort}(sel_i) = sortdesc \text{ and } sel_i \text{ access the } i\text{'th component.}\\
&TypeToContext\ [\bot] = \{[\ ]\}
\end{aligned}$$

Given the type and the final context of the k'th parameter of the function `f`, the context

$$TypeToContext\ [GenType\ [type(f,k)]\ cont(f,k)]$$

provides us with information about how the k'th parameter of `f` is to be split. Of course, this final context is not known. Therefore we have to find the context by a fixpoint iteration, as described in subsection 6.5 below.

In the above example only one function, `f`, is considered. In the case of a higher order functional language, given an application:

$$(exp_0\ \ exp_1 \ldots exp_n)$$

we have to take into account all functions that exp_0 may evaluate to.

The closure analysis provides us with a set of functions that exp_0 may evaluate to. Given this set, we can find the total context of the k'th parameter, by:

$$\bigcup \{ TypeToContext[GenType\ [type(f,k)]\ cont(f,k)]\},\ \texttt{where}\ f \in closure\text{-}set(exp_0)$$

i.e. the total context is all attempts at accessing the k'th parameter in the body of all lambda abstractions in the set returned by the function *closure-set*.

We only need one context for all lambda abstractions in a given closure-set. That is because we use the context information to decide how to split parameters and all functions in the closure-set must be split the same way. Therefore, instead of assigning a context to each lambda abstraction we only assign contexts to each closure-set.

The same can be done for the types of the parameters of all lambda abstractions in all closure-sets in the program by

$$type(cl,k) = \bigsqcup_{f \in cl} \{type(f,k)\},\ \text{k=1} \ldots arity(\text{f}),\ \text{for all closure-sets, } cl, \text{ of the program.}$$

Given an expression *exp* the function *closure-set-name*(*exp*) returns the name of the closure-set that *exp* is a member of.

We use the terms *cont*(cl,k) and *type*(cl,k) to denote the context and the type of the k'th parameter of members of the closure-set *cl*.

Given an application $(exp_0\ exp_1 \ldots exp_n)$ we can now find the closure-set, *cl*, that contains the lambda abstractions that exp_0 may evaluate to, and then the context

$$TypeToContext\ [GenType\ [type(cl,k)]\ cont(cl,k)]$$

provides us with a context that corresponds to the total context of the k'th parameter given above.

Therefore we get the function C listed below:

$$\texttt{C}\ x\ [\![y]\!]\ \Pi = \begin{cases} \Pi & \texttt{if } x = y \\ \{[\ ]\} & \texttt{otherwise} \end{cases}$$

$$\texttt{C}\ x\ [\![constant]\!]\ \Pi = \{[\ ]\}$$

$$\texttt{C}\ x\ [\![(\texttt{if}\ exp_0\ exp_1\ exp_2)]\!]\ \Pi = \bigcup_{i=0}^{2} \texttt{C}\ x\ [\![exp_i]\!]\ \{[\ ]\}$$

$$\texttt{C}\ x\ [\![(predicate\ exp_1 \ldots exp_n)]\!]\ \Pi = \bigcup_{i=1}^{n} \texttt{C}\ x\ [\![exp_i]\!]\ \{[\ ]\}$$

$$\texttt{C}\ x\ [\![(operator\ exp_1 \ldots exp_n)]\!]\ \Pi = \bigcup_{i=1}^{n} \texttt{C}\ x\ [\![exp_i]\!]\ \{[\ ]\}$$

$$\mathtt{C}\ x\ [\![(sel\ exp)]\!]\,\Pi = \mathtt{C}\ x\ [\![exp]\!]\left(\{[sel] \frown \pi \mid \pi \in \Pi\} \cup \{[\,]\}\right)$$

$$\mathtt{C}\ x\ [\![(constructor\ exp_1\ \ldots exp_n)]\!]\,\Pi = \bigcup_{i=1}^{n} \mathtt{C}\ x\ [\![exp_i]\!]\,\{[\,]\}$$

$$\mathtt{C}\ x\ [\![(\mathtt{lambda}\ (v_1\ \ldots v_n)\ exp)]\!]\,\Pi = \begin{cases} \{[\,]\} & \mathtt{if}\ x \in \{v_1 \ldots v_n\} \\ \mathtt{C}\ x\ [\![exp]\!]\,\{[\,]\} & \mathtt{otherwise.} \end{cases}$$

$$\mathtt{C}\ x\ [\![(\mathtt{let}\ ((name_1\ exp_1)\ \ldots (name_n\ exp_n))\ exp_0)]\!]\,\Pi = \bigcup_{i=1}^{n} \mathtt{C}\ x\ [\![exp_i]\!]\,\{[\,]\} \cup \begin{cases} \{[\,]\} & \mathtt{if}\ x \in \{name_1 \ldots name_n\} \\ \mathtt{C}\ x\ [\![exp_0]\!]\,\{[\,]\} & \mathtt{otherwise} \end{cases}$$

$$\mathtt{C}\ x\ [\![(exp_0\ exp_1\ \ldots exp_n)]\!]\,\Pi = \mathtt{C}\ x\ [\![exp_0]\!]\,\{[\,]\} \cup \bigcup_{i=1}^{n} \mathtt{C}\ x\ [\![exp_i]\!]\ cont(\mathrm{cl,i}),\ \mathtt{where}\ cl = closure\text{-}set\text{-}name(exp_0)$$

6.5 Finding the context by fixpoint iteration

Due to the latent selectors, we cannot find the context by only analyzing the body of each function. The type and context must be known, when finding the context. We therefore have to find the context iteratively.

Initially let all contexts be empty, i.e.

$$cont(cl,\mathrm{k}) = \{[\,]\},\ \mathtt{for\ k=1,\ \ldots,}\ arity(\mathrm{f})\ \mathtt{where}\ f \in cl$$

for all closure-sets, cl, in the program.

As mentioned above, we intend to find a new approximation to the final context by applying the function C to the body-expression of all functions and lambda abstractions, i.e.

$$cont(\mathrm{f,k}) = \mathtt{C}\ param(\mathrm{f,k})\ [\![body(\mathrm{f})]\!]\,\{[\,]\},\ \mathtt{for}\ k = 1,\ \ldots,\ arity(\mathrm{f})$$

Given the closures in the program we now have to find the context of all lambda abstractions in the closure-sets, i.e.

$$cont(cl,\mathrm{k}) = \bigcup_{f \in cl} cont(\mathrm{f,k})$$

However it leads to fewer calculations if we find the context by the following equations

$$cont(cl,\mathrm{k}) = TypeToContext\left[\ GenType\ [type(cl,\mathrm{k})]\ \bigcup_{f \in cl} \left\{\mathtt{C}\ param(\mathrm{f,k})\ [\![body(\mathrm{f})]\!]\,\{[\,]\}\right\}\ \right]_{k=1,\ldots,arity(f)}$$

We have now defined an iterative algorithm that can find the context of all parameters of all functions and lambda abstractions in the program.

We assume the set *Context* to be equipped with the natural partial ordering. Then the functions *TypeToContext*, *GenType* and C are monotonic with respect to contexts and therefore the minimal fixed point for the above equations exist. Given that $cont(cl,k) \subseteq$ *TypeToContext* [$type(cl,k)$] and *TypeToContext* [$type(cl$,k)] is a finite set, we can see, that only a finite number of contexts can be taken as value by $cont(cl$,k), and therefore the minimal fixed point can be found by a finite number of iterations.

Given the contexts we can now generalize the types of all parameters in the program, i.e. find new types, using the following equations

$$type'(cl,\mathrm{k}) = GenType[type(cl,\mathrm{k})]\ cont(cl,\mathrm{k}),\ \text{for } k{=}1 \ldots arity(cl)$$

for all closure-sets, cl, in the program. We are now ready to perform the actual splitting of the parameters in the program.

7 Using Type Information for Parameter Splitting

Let us start by taking a closer look at the type t of a parameter, x. A type can be viewed as a tree, with all its leaves being *any*, $atom(\mathcal{A})$, $lambda(\mathcal{L})$ or $\perp$, built using different constructors.

However, let us suppose that one of the leaves in the tree is $\perp$. This would mean that no S-expression could be taken as value of the leaf, and thereby of the variable. Therefore the function to which the parameter belongs would never be called, so it could just be removed from the program. Therefore we assume that no leaf is $\perp$.

Therefore the leaves can only be one of *any*, $atom(\mathcal{A})$ and $lambda(\mathcal{L})$. All leaves assigned to *any* will be referred to as *gaps*.

A leaf containing $lambda(\mathcal{L})$ can be regarded as a special constant. Therefore the lambda abstraction, $\mathcal{L}$, can be lifted from the application points into the body of the other lambda abstractions in the following way:

Example 11 lifting lambda abstractions
Consider the following program:

```
(loadt "scheme.adt")
(define (f x)
   (g (lambda (y) (cons y 2)) x))
(define (g fun x)
   (fun x))
```

The lambda abstraction occurring in the body of f can be lifted into the body of the function g. This will give the program:

```
(loadt "scheme.adt")
(define (f x)
   (g x)
(define (g x)
   ((lambda (y) (cons y 2)) x))
```

Lambda lifting can however give some problems. For instance it will not be correct to perform this simple lifting, if there are free variables in the lambda abstraction.

Due to the problems that might occur when lifting lambda abstractions into other functions we will not in this paper describe how to perform the lifting. Lifting can be prevented by generalizing all occurrences of $lambda(\mathcal{L})$ to *any*. In section 8 we will address some of the problems with lifting lambda abstractions and suggest how they can be solved.

It is obvious that given two variables x and y described by the same type t, their actual values can only differ at the places corresponding to the gaps, and must be congruent at all other places. Given a leaf denoted by $atom(\mathcal{A})$, both variables have the same atom as value of the leaf, the value being described by $\mathcal{A}$. The least upper bound of different atoms is *any*.

Assume that the type t contain m gaps. Any value described by the type, t, is completely determined by the type, t and the value of the parts corresponding to the gaps in the type. Therefore, it is reasonable to split a variable x of the type t into m new variables where each new variable correspond to a *gap* in the type tree.

We now define a function *CountGaps* that given a type, produces the natural number equal to the number of gaps in the type, i.e. the number of *any*-leaves in the type tree.

$$CountGaps \in Type \rightarrow \mathcal{N}$$

$$CountGaps[any] = 1$$
$$CountGaps[atom(\mathcal{A})] = 0$$
$$CountGaps[sortdesc(t_1, \ \ldots, \ t_m)] = CountGaps(t_1) + \ldots + CountGaps(t_m)$$

Let us now take a closer look at the parameter splitting method of section 3. Let the original formal parameter x_k be described by the type $sortdesc(t_k^{(1)}, \ t_k^{(2)}, \ \ldots, \ t_k^{(m)})$. Let $[x_k^{(1)} \ x_k^{(2)} \ \ldots \ x_k^{(m)}]$ be a list of m new variables. In the first step the original actual parameter, x_k, is replaced by the list of m new variables, and all occurrences of the parameter in the body of the function is replaced by $(constructor \ x_k^{(1)} \ x_k^{(2)} \ \ldots \ x_k^{(m)})$. We want to generalize this strategy.

Let us suppose that the parameter x_k has the type t and $m = CountGaps[t]$. We can easily get a list of m new variables $[x_k^{(1)} \ x_k^{(2)} \ \ldots \ x_k^{(m)}]$. Replacing the actual parameter x_k with the list of the m new parameters work as described above. All use of the original formal parameter x_k in the body of the function $body(f)$ must now be replaced by an expression synthesizing the value of the original parameter x_k from the values of the new parameters. To this purpose we define a function, *ExpandVar*, which given a type, t, and a list of m new variables, $[x_k^{(1)} \ x_k^{(2)} \ \ldots \ x_k^{(m)}]$, where $m = CountGaps[t]$, produces the expression required.

$$ExpandVar \in Type \rightarrow VName^* \rightarrow S\text{-}expression$$

$$ExpandVar\ [any]\ [x] = x$$
$$ExpandVar\ [atom(\mathcal{A})]\ [\ ] = quote\ \ \mathcal{A}$$

$$ExpandVar\,[sortdesc(t_1,\ \ldots,\ t_m)]\,X =$$
$$(constructor\ \ ExpandVar\,[t_1]\,X_1\ \ \ldots\ \ ExpandVar[t_m]X_m),\ \texttt{where}$$
$$X_1 \frown \ldots \frown X_m = X\ \texttt{and}\ sort(constructor)\ \texttt{=}\ sortdesc\ \texttt{and}$$
$$len(X_1) = CountGaps(t_1),\ \ \ldots,\ \ len(X_m) = CountGaps(t_m)$$

We have used the term $len(A)$ to denote the length of the list A.

We have now generalized the first step of the method described in section 3 to split a parameter to a function. Assuming that the function `f`

$$\texttt{(define (f } \ldots\ x_k\ \ldots\texttt{)}\ body_f\texttt{)}$$

is defined in the original program, we will replace this definition with

$$\texttt{(define (f } \ldots\ x_k^{(1)}\ x_k^{(2)}\ \ldots\ x_k^{(m)}\ \ldots\texttt{)}$$
$$(body(f)[x_k \rightarrow (ExpandVar\,[type(f,k)]\,[x_k^{(1)}\ x_k^{(2)}\ \ldots\ x_k^{(m)}]\,)\,]\,)\texttt{)}$$

The second step in the original method is to replace all applications of the function `f`

$$\texttt{(f } \ldots\ exp_k\ \ldots\texttt{)}$$

with a new application

$$\texttt{(f } \ldots\ (sel_1\ exp_k)\ (sel_2\ exp_k)\ \ldots\ (sel_m\ exp_k)\ \ldots\texttt{)}$$

This must also be generalized. We have to select the subparts of the expression exp_k that correspond to the new variables, i.e. the gaps in the type t. To this purpose we define a function, *SplitArg*, that given a type t and an expression exp produces a list of expressions corresponding to the new m formal parameters. This can be done because we for every sort know the selectors accessing each subpart of the constructed element.

$$SplitArg \in Type \times Exp \rightarrow Exp^*$$

$$SplitArg(any, exp) = [exp]$$
$$SplitArg(atom(\mathcal{A}), exp] = [\ ]$$
$$SplitArg(sortdesc(t_1,\ \ldots,\ t_m), exp) =$$
$$[SplitArg(t_1,\ selector_1(exp)\,)\,]\frown \ldots \frown [SplitArg(t_m,\ selector_m(exp)\,)\,]$$
$$\texttt{where}\ sort(selector_1)\ \texttt{=}\ \ldots\ \texttt{=}\ sort(selector_m)\ \texttt{=}\ sortdesc$$

We now have an algorithm that can be used to split parameters of all functions and lambda abstractions in the program.

During the splitting we may have introduced some selector-constructor combinations that may be reduced. Most of these combinations can be eliminated in a trivial way.

8 Lifting lambda abstractions

If a leaf of the type tree is marked by $lambda(\mathcal{L})$ then that part of the tree can almost be regarded as constant. If there are free variables occurring in the body of the lambda abstraction, then the free variables will have values depending on the environment in which the lambda abstraction was defined. If we want to do lambda lifting we must regard all free variables in lambda abstractions as gaps. The reason for this is illustrated by the following example:

Example 12 Lifting lambda abstractions with free variables
Consider the following program:

```
(loadt "scheme.adt")
(define (f x)
    (g (lambda (y)
          (cons x y))))
(define (g fun)
   (fun 3))
```

If we regard the lambda abstraction as a constant that may be freely copied, then we can get the following program when we split the parameters of the program:

```
(loadt "scheme.adt")
(define (f x) (g))
(define (g)
   ((lambda () (cons x 3))))
```

The resulting program is obviously incorrect. The variable x is not bound to anything in the body of the function g. One way of solving this problem is to propagate the free variables in the lambda abstraction to the function g. If we do this we get the program:

```
(loadt "scheme.adt")
(define (f x) (g x))
(define (g x)
   ((lambda () (cons x 3))))
```

We thus see that we have to propagate the free parameters of lambda abstractions to the functions where the lambda abstractions are being used.

Lifting anything else than name-less lambda abstractions can be done without considering free variables since there are none.

There are other problems associated with lambda lifting (e.g. what about lifting a lambda abstraction into itself). We will consequently not describe the whole theory here.

9 Summary

In order for the result produced by parameter splitting to be reasonable, we need information obtained by two preliminary global analyses of the program. The first, forward, analysis tells us whether the splitting is feasible, whereas the second, backward, analysis tells us whether the splitting is useful.

The first, forward, analysis result in the parameters of the functions being assigned *types*, which describe the structure of the argument expressions in the function calls.

The second, backward, analysis results in the parameters of the functions being assigned *contexts*, which provide information about attempts at accessing the parameters.

Then the information obtained is used to perform variable splitting. The type information is used to avoid introducing new selectors into the program as well as code duplication, whereas the context information makes it possible to avoid useless parameter splitting that does not cause some selectors in the program to be eliminated.

References

[Bondorf 90a] Bondorf, A. Automatic Autoprojection of Higher Order Recursive Equations. In Jones, Neil D. (editor), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 70–87. Springer-Verlag, May 1990.

[Bondorf 90b] Bondorf, Anders and Danvy, Olivier. *Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types.* Technical Report 90/4, DIKU, University of Copenhagen, Denmark, 1990.

[Gomard 89] Gomard, C. K. Higher Order Partial Evaluation – HOPE for the Lambda Calculus. Master's thesis, DIKU, University of Copenhagen, Denmark, September 1989.

[Gomard 90] Gomard, Carsten K. Partial Type Inference for Untyped Functional Programs. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, ACM, pages 282–287. 1990.

[Jones 85] Jones, N.D., Sestoft, P., and Søndergaard, H. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In Jouannaud, J.-P. (editor), *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, Vol. 202)*, pages 124–140. Springer-Verlag, 1985.

[Jones 89] Jones, N.D., Sestoft, P., and Søndergaard, H. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation, *Lisp and Symbolic Computation* 2(1):9–50, 1989.

[Jones 90] Jones, N.D., Gomard, C.K., Bondorf, A., Danvy, O., and Mogensen, T.Æ. A Self-Applicable Partial Evaluator for the Lambda Calculus. In *1990 International Conference on Computer Languages, New Orleans, Louisiana, March 1990*, IEEE Computer Society, pages 49–58. 1990.

[Mogensen 89] Mogensen, Torben Ægidius. *Binding Time Aspects of Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, March 1989. 95 pages.

[Romanenko 88] Romanenko, S.A. A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure. In Bjørner, D., Ershov, A.P., and Jones, N.D. (editors), *Partial Evaluation and Mixed Computation*, pages 445–463. North-Holland, 1988.

[Romanenko 90] Romanenko, Sergei A. Arity Raiser and Its Use in Program Specialization. In Jones, N. (editor), *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Springer-Verlag, 1990.

[Schmidt 86] Schmidt, David A. *Denotational Semantics, a Methology for Language Development.* Allyn and Bacon, Boston, 1986.

[Sestoft 86] Sestoft, P. The Structure of a Self-Applicable Partial Evaluator. In Ganzinger, H. and Jones, N.D. (editors), *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, Vol. 217)*, pages 236–256. Springer-Verlag, 1986.

[Sestoft 88] Sestoft, Peter. Replacing Function Parameters by Global Variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988.