Binding Time Aspects of Partial Evaluation

Torben Ægidius Mogensen
DIKU
University of Copenhagen
Denmark

3 March 1989

Contents

Chapter 1

Overview

This thesis is the outcome of studies towards the Danish "licentiatgrad", the Danish equivalent of a Ph.D. degree. The topic of the thesis is partial evaluation and in particular how explicit treatment of binding times affects generation of compilers by self-application of partial evaluators.

In chapter 2 we start with the concept of and some theory regarding partial evaluation. In chapter 3 we continue with a presentation of various algorithms for partial evaluation and examine one of these in detail. Self-application of this algorithm for the purpose of compiler generation is investigated in chapter 4, where it is concluded that explicit treatment of binding times is needed, leading to a variant of the algorithm. In chapter 5 new versions of the equations from chapter 2 are shown.

The following three chapters: 6, 7 and 8 present various algorithms for binding time analysis and describe how they can be used in partial evaluator systems. These chapters are modified versions of articles that were produced during the period of the studies. Chapter 6 corresponds to [Mogensen 88], chapter 8 corresponds to [Mogensen 89b] and the contents of chapter 7 have been submitted for publication.

Chapter 9 presents a collection of ideas related to the subject of the thesis.

Some of the chapters or sections contain conclusions, but a short general conclusion is given in chapter 10.

Acknowledgements

My thanks must primarily go to Neil D. Jones, who has been my supervisor for this thesis. The next in line is Chris Hankin of Imperial College who supervised me when I was a visiting student at Imperial College in the summer of '88. The work that resulted in chapter 8 was primarily done at Imperial College.

All the people in the MIX group at DIKU have been helpful in the production of this thesis by giving helpful comments about early versions of this work, by participating in discussions and giving general moral support. The very special research climate in this group has been of paramount importance for this work.

During my visits to Imperial College and Glasgow University and at the Workshop of Partial Evaluation and Mixed Computation in Denmark in 1987 I met a multitude of people with whom I had lengthy discussions about partial evaluation. Of these, in particular John Launchbury from Glasgow has provided useful insights and inspiration.

Chapter 2

Introduction

This chapter presents the basic concept and theory of partial evaluation, as found in [Ershov 82], [Jones *et al.* 85] or [Jones *et al.* 88]. This is followed by a section that presents the main thesis of this thesis: when self-applying a partial evaluator to generate compilers or other program generators, it is necessary to handle binding time information explicitly, preferably in the form of a preliminary binding time analysis of the program that later will be partially evaluated. This will then lead naturally to the following chapters that describe ways of doing binding time analysis and how to use the information obtained.

2.1 Partial Evaluation

Partial evaluation is a way of executing programs, even if only part of the input to the program is present. Obviously, this will in general not lead to the output of a final answer, as this answer is likely to depend on the remaining input. What *can* be done is, however, producing a new *residual* program which, when given the remaining input, will produce the final answer. By returning programs, partial evaluation have some similarities to program transformation. But since program transformations normally don't use any of the input, partial evaluation is something in between normal evaluation and program transformation.

This concept is familiar from functional programming where it corresponds to doing a partial application, which results in an object that when applied to the remaining arguments will produce the result. In functional programming this object is, however, normally a *suspension* leaving all the real work to be done when the remaining arguments are supplied. In partial evaluation the point is *not* to postpone the work: we want as much as possible done at partial evaluation time, so the work left to do when the remaining arguments are supplied is as little as possible. When we in this chapter present the basic theory, it is worth to note that the equations state nothing about the amount of work done at partial evaluation time; this is left to more informal speculations in the following chapters.

2.2 Some theory of partial evaluation

In this section we present some notation and theory which will be used later.

2.2.1 Languages, interpreters and compilers

We will use a notation similar to [Jones *et al.* 85] and [Jones *et al.* 88], where a language is represented by a semantic function: a partial function from a domain of program representations (syntax) to a domain of partial functions over the value domain of the language:

$$\mathbf{L} : P \rightarrow D \rightarrow D$$

Thus $\mathbf{L}\ p$ represents the input/output function of the **L**-program p and $\mathbf{L}\ p\ d$ is the result of executing the program p with the input d.

We will from now on assume that the domain of program representations P is a subset of the domain of values D, thus allowing programs as data objects. We will assume that the data domain allows pairing of any values: $(d_1, d_2) \in D$ if $d_1 \in D$ and $d_2 \in D$. Note that having programs as data objects is not the same as having functions as data objects as in higher order languages. The programs have no inherent meaning, they are just "texts" that can be examined, built and decomposed.

An interpreter for the language **S** written in **L** is an **L**-program *s_int* with the following property:

$$\mathbf{L}\ s\_int\ (p, d) = \mathbf{S}\ p\ d$$

for all **S**-programs p and input d. We use $=$ to mean that either both values are undefined, or they are both defined and equal.

A compiler *comp* written in **L**, compiling **S**-programs to **T**-programs is defined by the equation:

$$\mathbf{T}\ (\mathbf{L}\ comp\ p)\ d = \mathbf{S}\ p\ d$$

for all **S**-programs p and input d.

2.2.2 Residual programs and partial evaluators

We now define a *residual program* of an **L**-program p with respect to a partial input d_1 to be any **L**-program p_{d_1} such that:

$$\mathbf{L}\ p_{d_1}\ d_2 = \mathbf{L}\ p\ (d_1, d_2)$$

for all input d_2. In this definition we assume that p has a pair of values as input, and that the first element of this is given as partial input with the second being the remaining input. The concept of a residual program can easily be generalized to other divisions of the input, but as this becomes notationally messy we stick to this simple definition for now.

In a way p_{d_1} is a version of p, *specialized* with respect to a constant first argument d_1.

A *partial evaluator* is a program *mix* that given a program p and data d_1 produces a residual program p_{d_1}:

$$\mathbf{L}\ mix\ (p, d_1) = p_{d_1}$$

or, combining the two definitions into what is known as the *mix equation*:

$$\mathbf{L}\ (\mathbf{L}\ mix\ (p, d_1))\ d_2 = \mathbf{L}\ p\ (d_1, d_2)$$

for all L-programs p and all data d_1, d_2. The data d_1 that is given to *mix* is called the *static data* of p, and the remaining data d_2 that is given to the residual program is the *dynamic data.* Note that there may be several equivalent residual programs p_{d_1} for any given p and static data d_1. Different partial evaluators can produce different residual programs, some more efficient than others. In the worst case very little is done at partial evaluation time, effectively suspending all of p's calculations until the dynamic input is given. For some problems this is all one can hope for, as the calculations can be heavily dependent on the dynamic input, but the applications shown below would be of little interest if we couldn't do a substantial part of the work at partial evaluation time.

2.2.3 Compiling and compiler generation

Using the mix equation with p being an interpreter *s_int* and d_1 being an S-program *source* we get:

$$\mathrm{L}\ (\mathrm{L}\ mix\ (s\_int, source))\ d\ =\ \mathrm{L}\ s\_int\ (source, d)$$

so s_int_{source} is an L-program that given d produces the same result as the S-program *source* does given the same input. s_int_{source} is thus *source* compiled from S to L:

$$object\ =\ \mathrm{L}\ mix\ (s\_int, source)$$

Which is commonly known as *the first Futamura projection*, due to its initial presentation in [Futamura 71]. The *second Futamura projection* states that a compiler may be generated by self-applying a partial evaluator:

$$comp\ =\ mix_{s\_int}\ =\ \mathrm{L}\ mix\ (mix, s\_int)$$

where

$$\mathrm{L}\ (\mathrm{L}\ comp\ source)\ d\ =\ \mathrm{S}\ source\ d$$

So $comp\ =\ mix_{s\_int}$ is an S to L compiler written in L. The second Futamura projection can easily be verified by:

$$object\ =\ \mathrm{L}\ mix\ (s\_int, source)\ =\ \mathrm{L}\ mix_{s\_int}\ source\ =\ \mathrm{L}\ comp\ source$$

Even though we, by using partial evaluation as in the first Futamura projection, can compile, it is interesting to have stand-alone compilers, as using these can be more efficient. For similar reasons we can use a compiler generator to generate compilers:

$$comp\ =\ \mathrm{L}\ cogen\ s\_int$$

The compiler generator *cogen* can be generated by *the third Futamura projection*:

$$cogen\ =\ mix_{mix}\ =\ \mathrm{L}\ mix\ (mix, mix)$$

Which can be verified by:

$$comp\ =\ mix_{s\_int}\ =\ \mathrm{L}\ mix\ (mix, s\_int)\ =\ \mathrm{L}\ mix_{mix}\ s\_int\ =\ \mathrm{L}\ cogen\ s\_int$$

In these equations we assume that *mix* is an *autoprojector*: the language it is written in is the same as the language it handles; the language of both of the programs it accepts as input and the residual programs it outputs. For self-application in the second and third Futamura projections it is essential that it is written in the language it accepts as input. The residual programs may, however, be in another language. This possibility is investigated in [Holst 88], where the residual programs are in a lower level language than the input programs. This results in compilers that are written in and outputs object programs in this low level language.

2.3 Other uses of partial evaluation

Partial evaluation can be used for other things than compiling and compiler generation. If we instead of an interpreter use a universal parser, *i.e.*, a program that given a grammar and a string parses the string according to the grammar. Partially evaluating the universal parser with respect to the grammar yields a parser for that grammar. Using the second Futamura projection we obtain a parser generator: a program that given a grammar produces a parser for that grammar. This is investigated in [Dybkjær 85]. In [Mogensen 86], partial evaluation is used to specialize a ray-tracer to a particular scene description, leaving the view-point as dynamic input.

Chapter 3

Partial Evaluation Algorithms

In this chapter we present various algorithms for partial evaluation of a small functional language. One of these algorithms is examined in closer detail.

3.1 A language

When describing the algorithms we will use a simple language of first order recursion equations. Throughout most of the thesis, variants of this language will be used.

A program consists of a set of function definitions:

$$
\begin{array}{lll}
program & ::= & f_1(x_{11} \ldots x_{1n_1}) = exp_1 \\
 & & \ldots \\
 & & f_m(x_{m1} \ldots x_{mn_m}) = exp_m \\
 & & \\
exp & ::= & x_{ij} \\
 & | & (QUOTE\ constant) \\
 & | & (base_k\ exp \ldots exp) \\
 & | & (IF\ exp\ exp\ exp) \\
 & | & (CALL\ f_i\ exp \ldots exp)
\end{array}
$$

Each function is defined by an expression that is built from constants, conditionals, function calls and uses of base functions. We will not fix the set of base functions. As can be seen from the syntax, we use a LISP-like notation. This is convenient when using programs as data objects. The first function definition defines the goal function: the function that is called when the program is executed. We assume a LISP-like call-by-value semantics, but most of the discussion in the following sections would apply equally well to call-by-name (or lazy) semantics. As an example program, the *append* function is presented below:

$$
\begin{array}{l}
append(a\ b) = \\
\quad (IF\ (NULL\ a)\ b\ (CONS\ (CAR\ a)\ (CALL\ append\ (CDR\ a)\ b)))
\end{array}
$$

3.2 Five algorithms for partial evaluation

In this section we describe five different algorithms for performing partial evaluation. We start by very simple, but also fairly uninteresting algorithms and work up to more complex and more interesting methods.

3.2.1 Algorithm 1: constant argument

This is the simplest possible partial evaluation algorithm: an extra function definition is added, calling the original goal function with constant arguments. As an example, partially evaluating the *append* function with $a = (1\ 2)$ as static data yields:

$$append_{(1\ 2)}(b) = (CALL\ append\ (QUOTE\ (1\ 2))\ b)$$
$$append(a\ b) =$$
$$(IF\ (NULL\ a)\ b\ (CONS\ (CAR\ a)\ (CALL\ append\ (CDR\ a)\ b)))$$

Thus all calculation is postponed until the dynamic argument b is supplied. This algorithm corresponds closely to building a closure, but is of little practical interest. It does, however, constitute a proof of existence of a partial evaluation algorithm. It was used as such in the proof of Kleenes *s-m-n theorem*, which essentially states the existence of a (primitive recursive) function *s-m-n* for any positive integers m and n, which will partially evaluate any function with $m + n$ arguments, given the definition of the function and the values of the m first arguments.

3.2.2 Algorithm 2: constant folding

This algorithm uses the constant folding principle known from compilation [Aho *et al.* 86]: if a variable has a known constant value throughout all possible executions, replace all occurrences of the variable with the constant and remove the variable from the parameter list. If the static parameters are not constant throughout the computation, the method from algorithm 1 is used. This gives the same residual program as above when specializing *append* with a static first argument, but if the second argument, b, is static (with the value (1 2)), we obtain:

$$append_{(1\ 2)}(a) =$$
$$(IF\ (NULL\ a)\ (QUOTE\ (1\ 2))\ (CONS\ (CAR\ a)\ (CALL\ append_{(1\ 2)}\ (CDR\ a))))$$

This algorithm will, as algorithm 1, always terminate. It is, however, inadequate for all but the simplest purposes, in particular it will be no better than algorithm 1 for compiling by specializing an interpreter.

3.2.3 Algorithm 3: constant folding with function call unfolding

By combining constant folding with unfolding of recursive function calls that are entirely controlled by static variables, we can get better results. Seeing that the recursion in *append* is controlled by a, we can get the following residual program by specializing *append* to $a = (1\ 2)$:

$$append_{(1\ 2)}(b) = (CONS\ (QUOTE\ 1)\ (CONS\ (QUOTE\ 2)\ b))$$

which can not be optimized further. The condition that the recursion must be entirely under static control is a bit vague, but can be given precise definitions. If a sufficiently strong criteria is chosen, unfolding will always be finite, so the algorithm will terminate.

This algorithm is, like the previous two, a *monovariant* algorithm, a term defined by Bolyonkov ([Bulyonkov 84], [Bulyonkov 88]) to mean that any residual program produced by the algorithm will contain only *one* specialized version (variant) of each of the functions in the original program. This means that the number of functions in the residual program is limited by the number of functions in the original program. If the original program is an interpreter, we will thus never get object programs containing more functions than the interpreter *no matter*

how large and complex the source program is. This leads to the conjecture that the object programs will have an unnatural (and inefficient) structure. This conjecture is supported by past experience.

3.2.4 Algorithm 4: polyvariant specialization

Polyvariant partial evaluation [Bulyonkov 84], [Bulyonkov 88] is similar to constant folding, as it replaces static variables by their values. The difference is that it does *not* require the value to be the same throughout the computation: if a different value is obtained, a new variant of the function that uses the variable is made, specialized with respect to this new value. Specializing *append* with $a = (1\ 2)$ thus yields:

$$\begin{aligned} append_{(1\,2)}(b) &= (CONS\,(QUOTE\,1)\,(CALL\,append_{(2)}\,b))) \\ append_{(2)}(b) &= (CONS\,(QUOTE\,2)\,(CALL\,append_{()}\,b))) \\ append_{()}(b) &= b \end{aligned}$$

Polyvariant specialization can be combined with unfolding of function calls, yielding a residual *append* program similar to the one from algorithm 3. Note that unfolding can be applied *after* specialization without changing the resulting residual program. This algorithm is (even in the absence of unfolding) *not* guaranteed to terminate: an infinite number of variants can be made if a static variable can obtain an infinite number of values. An example of this is the function:

$$f(a\ b) = (IF\,(>\ a\ b)\,a\,(CALL\,f\,(+\ a\ a)\,b))$$

which when specialized to $a = 1$ yields the infinite sequence:

$$\begin{aligned} f_1(b) &= (IF\,(>\ (QUOTE\,1)\,b)\,(QUOTE\,1)\,(CALL\,f_2\,b)) \\ f_2(b) &= (IF\,(>\ (QUOTE\,2)\,b)\,(QUOTE\,2)\,(CALL\,f_4\,b)) \\ f_4(b) &= (IF\,(>\ (QUOTE\,4)\,b)\,(QUOTE\,4)\,(CALL\,f_8\,b)) \\ &\ldots \end{aligned}$$

To ensure termination of polyvariant specialization it is thus necessary to find a way of limiting the number of variants. A fixed limit is inappropriate, as this will impose similar restrictions on residual programs as monovariant specialization. [Jones 88] discusses this problem in details.

An example where polyvariant specialization yields a better result than monovariant specialization is Ackermann's function:

$$\begin{aligned} &ack(a\ b) = \\ &\quad (IF\,(=\ a\ 0)\,(+\ b\ 1) \\ &\qquad (IF\,(=\ b\ 0)\,(CALL\,ack\,(-\ a\ 1)\ 1) \\ &\qquad\quad (CALL\,ack\,(-\ a\ 1)\,(CALL\,ack\,a\,(-\ b\ 1))))) \end{aligned}$$

where we have omitted *QUOTE* on numeric constants for readability.

The recursion is not controlled entirely by any one of the parameters, so algorithm 3 can not unfold the function calls. Neither is any variable constant, so a trivial result like in algorithm 1 is obtained. Using polyvariant specialization with respect to $a = 2$ we get:

$$\begin{aligned} &ack_2(b) = \\ &\quad (IF\,(=\ b\ 0)\,(CALL\,ack_1\ 1)\,(CALL\,ack_1\ (CALL\,ack_2\,(-\ b\ 1)))) \\ &ack_1(b) = \\ &\quad (IF\,(=\ b\ 0)\,(CALL\,ack_0\ 1)\,(CALL\,ack_0\ (CALL\,ack_1\,(-\ b\ 1)))) \\ &ack_0(b) = (+\ b\ 1) \end{aligned}$$

which can be improved further by unfolding the calls where the value of b is a constant.

Experience shows that polyvariant specialization is powerful enough to get good object programs when compiling by specializing interpreters [Jones *et al.* 85], [Jones *et al.* 88], chapter 6, [Bondorf 89], [Romanenko 88]. If the interpreter is interpreting the language in which it is written (it is a metacircular interpreter), object programs structurally identical to the source programs can be obtained. The partial evaluators presented in this thesis are all polyvariant specializers.

3.2.5 Algorithm 5: driving

There are even more powerful algorithms than polyvariant specialization. Turchin's supercompiler [Turchin 79], [Turchin 82], [Turchin 86a], [Turchin 86b] and Wadler's treeless transformation [Wadler 88] are based on *driving*. Where polyvariant specialization specializes functions with respect to known *values* of some of the parameters, driving can also specialize with respect to certain patterns of argument *expressions*, including calls to other functions. This is done by defining certain *configurations*; patterns matching certain expressions. By a combination of unfolding of functions and instantiation of variables, expressions are transformed to match one of these configurations, which is then replaced by a call to a residual function defined in terms of the configuration. As an example, using the configurations $(CALL\ append\ (CALL\ append\ X\ Y)\ Z)$ and $(CALL\ append\ X\ Y)$ where X, Y, Z are required not to contain function calls, we can obtain the following residual program for $(CALL\ append\ (CALL\ append\ a\ b)\ c)$:

$$
\begin{array}{l}
append3(a\ b\ c)\ = \\
\quad (IF\ (NULL\ a)\ (CALL\ append\ b\ c) \\
\qquad (CONS\ (CAR\ a)\ (CALL\ append3\ (CDR\ a)\ b\ c))) \\
append(a\ b)\ = \\
\quad (IF\ (NULL\ a)\ b\ (CONS\ (CAR\ a)\ (CALL\ append\ (CDR\ a)\ b)))
\end{array}
$$

The details of how this is done are omitted, see [Turchin 79], [Turchin 86b] or [Wadler 88]. Termination of driving is even harder to guarantee than termination of polyvariant specialization. Infinite variants can be obtained or the transformation may never reach one of the configurations. Turchin handles the problem by expanding the set of configurations on-line when transformations seem to lead nowhere and by combining sets of configurations by generalization when infinite variants threaten. Wadler uses a restricted language subset, where transformation to a finite set of configurations is guaranteed.

3.2.6 Algorithms for other languages

Variants of these algorithms can be used for other languages. If a flow-chart language is used, think of basic blocks as functions and jumps as function calls. This corresponds closely to tail-recursive function calls with the entire store as parameters.

In logic languages, predicates correspond to functions. and solving of sub-goals to function calls. Backtracking and backwards unification makes book-keeping more difficult, but otherwise the above algorithms can be used in logic languages too.

3.3 Polyvariant Specialization

As mentioned above, we will use the polyvariant specialization algorithm to implement our partial evaluators. In this section will give a more detailed description of this algorithm.

3.3.1 Fixed point iteration

When doing polyvariant specialization, the residual program is gradually build up from specialized versions of the original functions. This is continued until all functions that are used are also defined, yielding a kind of fixed point. This algorithm is best described in imperative style using while-loops:

```
defined_funs := emptyset;
needed_funs := {(goal_function,static_pars)};
while needed_funs ≠ ∅ do begin
    (function,static_args) := select(needed_funs);
    needed_funs := needed_funs\{(function,static_args)};
    if (function,static_args) is_not_defined_in defined_funs then begin
        (arg_names,body) := definition_of function ;
        residual_body := specialize_expression(body,arg_names,static_args);
        defined_funs := defined_funs ∪ {(function,static_args,residual_body)};
        needed_funs := needed_funs ∪ functions_used_in(residual_body);
    end
end
```

We start by an empty set of defined residual function definitions, and a set of needed function descriptions, initially containing only a description of the intended goal-function of the residual program. Each description consists of a name of a function from the original program and values for the static parameters to this. The loop proceeds to generate residual definitions for the descriptions in needed_funs, adding these to defined_funs and searching the body expression for used function calls, adding these to needed_funs. The loop proceeds until there are no more needed functions.

The defined residual functions are represented by the name of the original function, the values of the static parameters and a specialized body expression. In a post-process these descriptions are converted to definitions by assigning each (function name / static values) pair an atomic name, using the dynamic parameter names as the new parameter list and the specialized expression as body. All calls in these body expressions are converted to use the atomic names for the residual functions rather than the (name / static values) pairs.

As an example, at entry to the algorithm needed_funs could be $\{(ack, (2\ *))\}$, where the $*$ marks a dynamic value. Assuming *ack* is defined as in section 3.2.4, defined_funs contains on exit:

$$\{(ack, (2\ *), (IF\ (=\ b\ 0)\ (CALL\ (ack, (1\ *))\ 1)\ (CALL\ (ack, (1\ *))\ (CALL\ (ack, (2\ *))\ (-\ b\ 1))))),$$
$$(ack, (1\ *), (IF\ (=\ b\ 0)\ (CALL\ (ack, (0\ *))\ 1)\ (CALL\ (ack, (0\ *))\ (CALL\ (ack, (1\ *))\ (-\ b\ 1))))),$$
$$(ack, (0\ *), (+\ b\ 1))\}$$

which is converted to the residual program shown in section 3.2.4.

3.3.2 Specialization of expressions

In the previous section we showed the central loop of a polyvariant specialization algorithm, which used a few procedures that was not defined. Of these, the only non-trivial one was specialize_expression, which specializes an expression with respect to an environment, where some of

the variables (the static) are given values and others (the dynamic) are not. We show the definition of such a function below. Both static and dynamic values are represented by expressions: static values are constant expression and dynamic expressions are arbitrary other expressions. Initially the values of the dynamic variables are variable names (as expressions), usually their own names. Thus the list (2 *) from the example above becomes the list ((*QUOTE* 2) *b*). The main idea of the algorithm is to evaluate base operators if all arguments (partially) evaluates to constant expressions, producing a new constant expression as the result. If any of the arguments to a base operator are non-constant, the resulting residual expression is constructed from the operator and the arguments. For simplicity, we represent the environment as separate name and value lists. We use symbols in italic capital font (like *CONS*) as syntax builders, both in patterns and expressions. So, if x is bound to 3 and y to 4, (*CONS* x y) will be the list (cons 3 4), and the pattern (*CAR* x) will match (car 3), (car (cons a b)) *etc.*

```
specialize_expression(exp,names,values) =
 case exp of
   atomic(name) :
       lookup(name,names,values)                    ; variable
   (QUOTE c) :
       exp                                          ; constant expression
   (CAR e) :
       case specialize_expression(e,names,values) of
         (QUOTE c) :
            (QUOTE car(c))
         re :
            (CAR re)
   (CDR e) :
       case specialize_expression(e,names,values) of
         (QUOTE c) :
            (QUOTE cdr(c))
         re :
            (CDR re)
   (CONS e1 e2) :
       let re1 = specialize_expression(e1,names,values) ,
           re2 = specialize_expression(e2,names,values)
       in case (re1,re2) of
           ((QUOTE c1),(QUOTE c2)) :
             (QUOTE cons(c1,c2))
           (re1,re2) :
             (CONS re1 re2)


                                                    ; further base operators similarly
```

```
(IF e1 e2 e3) :
    case specialize_expression(e1,names,values) of
      (QUOTE c) :                                    ; static condition
          if c then specialize_expression(e2,names,values)
          else specialize_expression(e3,names,values)
      re :                                           ; dynamic condition
          (IF re specialize_expression(e2,names,values)
                 specialize_expression(e3,names,values))
(CALL f e1 ... en) :
    let args = list(specialize_expression(e1,names,values),...,
                    specialize_expression(en,names,values))
    in
                  if the call should be unfolded then
                    let (par_names,body) = definition_of(f)
                    in
                       specialize_expression(body,par_names,args)
                  else
                    let sargs = replace_dynamic_by_*(args),
                        dargs = skip_static(args)
                    in
                        (CALL (f,sargs) . dargs)
```

If a call is unfolded, we perform a beta-reduction: we bind the formal parameters to the residual expressions of the actual parameters and specialize the body expression in this environment. If we don't unfold, we just build a call to a residual function by pairing the static arguments with the function name to construct the residual function identifier and use the dynamic arguments as arguments of the residual call. No rules to decide when to do or not to do unfolding are specified in the function; we will discuss some possibilities below.

Some extra features, like reduction rules to locally reduce residual expressions (such as (*CAR* (*CONS* a b)) = a) can be added.

3.3.3 Unfolding

As mentioned in section 3.2.4, unfolding can be postponed till after specialization. There are, however, good reasons to unfold during specialization: If the result of a function is static (constant), the value can only be used if the call is unfolded. Also, unfolding as a post-process is inefficient and can potentially take much more time than the actual specialization. Unfolding during specialization can on the other hand lead to infinite unfolding, so some care must be taken in deciding when to unfold. A possible solution is to use a safe but conservative unfolding strategy during specialization, and then use a post-process to do extra unfolding afterwards. This strategy is used in [Sestoft 88].

It is reasonable and mostly safe to unfold if all parameters to a function call are static. Reasonable, as the result is guaranteed to be static, if any result exist. Safe, since all recursion will be decided by static values, but as non-termination can still occur, only mostly safe. Additional unfolding is essential for a good result, but it is difficult to decide locally when unfolding will be safe. So, either a track of function calls must be kept during specialization, or else a global static analysis of the program must be done before specialization. The first strategy is

used in [Turchin 79], [Turchin 86b], where each function call is compared to previous calls, and suspended if it is similar to any of these. This is very time and space consuming, and for large programs this is where the system uses the majority of its time. If the result of a static analysis is used to annotate (mark) calls that can be safely unfolded during specialization, virtually no time is used during specialization. Some time is used prior to specialization to perform the static analysis, but the result can be used in subsequent specialization of the same program to different static values. This strategy is used in [Sestoft 88], and will be discussed further in the next section.

Chapter 4

Self-application and Binding Time Analysis

We will in this section investigate self-application of the specialize_expression function from section 3.3. We will see that this will yield compilers that are "vaster than empires, and more slow" [Marvell]. As a solution to this problem we introduce *binding time analysis*, a static analysis of the program that is to be partially evaluated. In addition to solving this problem, binding time analysis will be shown useful for, among other things, deciding when to unfold calls.

4.1 Binding time

We will the term *binding time* to describe the time at which the value of a variable or function is available. This is known from compilation, where it is normal to say that values available at compile time (such as types in Pascal) have *static* binding time, whereas run time values have *dynamic* binding ime. The terms *static scoping* and *dynamic scoping* reflect this.

When using a partial evaluator to compile, the values available during specialization correspond to compile time values, and are thus termed *static*. Similarly, the values that are not available until the residual program is executed are termed *dynamic*. We will use the term *meta-static* for values that are available at self-application time. In terms of compiler generation, the interpreter is meta-static, the source program is static and the input to the (source or object) program is dynamic.

4.2 Specialization

Consider this definition, which may be part of an interpreter:

```
(lookup (n ns vs) = (IF (EQUAL n (CAR ns))
                        (CAR vs)
                        (CALL lookup n (CDR ns) (CDR vs))
)                   )
```

When compiling by specializing an interpreter ($object = \mathbf{L}\ mix\ (int, p)$), we might specialize this function with n,ns being static and vs dynamic. This involves a call like:

```
specialize_expression((IF ...),(n ns vs), ((QUOTE y) (QUOTE (x y z)) vals))
```

Where the first parameter is the body expression of lookup. Assuming the decision procedure in specialize_expression is able to recognize that unfolding of the recursive calls in lookup is safe, we get the following residual expression:

(*CAR* (*CDR* vals))

This is no surprise, and not particularly interesting. Consider, however, compiler generation (*comp* = L *mix* (*mix*, *int*)). This involves *specializing* the call:

specialize_expression((*IF* ...),(n ns vs),*)

meaning, we do this call:

specialize_expression(*body of specialize_expression*, (exp names values), ((*QUOTE* (*IF* ...)) (*QUOTE* (n ns vs)) values))

The body of specialize_expression is in a "syntactically sugared" form of the language it specializes. It is, however, not difficult to apply the principles of the specialization to this extended language. Doing so, we get the residual function below, where we have the same notation as we did when describing specialize_expression:

```
specialize_expression(QUOTE (IF ...)),(QUOTE (n ns vs)) (values) =
   case
      let re1 = car(values),
          re2 = case car(cdr(values)) bf of
                      (QUOTE c) : (QUOTE car(c))
                      re        : (CAR re)
      in case (re1,re2) of
               ((QUOTE c1),(QUOTE c2)) : (QUOTE equal(c1,c2))
               (re1,re2)               : (EQUAL re1 re2)
   of
      (QUOTE c) :
         if c then case car(cdr(cdr(values))) of
                        (QUOTE c) : (QUOTE car(c))
                        re        : (CAR re)
         else let args = list(car(values),
                              case car(cdr(values)) of
                                   (QUOTE c) : (QUOTE cdr(c))
                                   re        : (CDR re)
                              case car(cdr(cdr(values))) of
                                   (QUOTE c) : (QUOTE cdr(c))
                                   re        : (CDR re)
                              )
              in
               if the call should be unfolded then
                  specialize_expression(QUOTE (IF ...)),(QUOTE (n ns vs))(args)
               else
```

```
                let sargs = replace_dynamic_*(args)
                    dargs = skip_static(args)

                in
                   (CALL ('lookup,args) . dargs)
re :
   (IF re
      case car(cdr(cdr(values))) bf of
           (QUOTE c) : (QUOTE car(c))
           re        : (CAR re)
      let args = list(car(values),
                      case car(cdr(values)) bf of
                           (QUOTE c) : (QUOTE cdr(c))
                           re        : (CDR re)
                      case car(cdr(cdr(values))) bf of
                           (QUOTE c) : (QUOTE cdr(c))
                           re        : (CDR re)
                      )

      in
         if the call should be unfolded then
            specialize_expression(QUOTE (IF ...),(QUOTE (n ns vs))(args)
         else
            let sargs = replace_dynamic_by*(args)
                dargs = skip_static(args)
            in
               (CALL ('lookup,args) . dargs)
      )
```

We assume that the lookup function used in specialize_expression is defined in the same way as the one we specialize, and have specialized the calls to it accordingly, yielding residual expressions like car(cdr(values)). We have unfolded the recursive calls to specialize_expression, except the one that specializes the body of the recursive call to lookup. We have left the unfold decision procedure in the residual program, as we assume that it is of the type that looks at a trace of previous calls, and as such will not have the necessary information available at compiler generation time.

It is immediately obvious that this residual function is quite large. There are several reasons for this. One is that the two branches of the case expression that handles the if expression contain a large amount of identical code, but even if this was solved, *e.g.*, by using a let construction, the residual function will still be large. A more fundamental reason can be recognized by considering the binding times of the variables in lookup. When we specialized lookup, we assumed that the name and name list (n,ns) were static and the value list (vs) was dynamic. This is quite natural, and we would expect a compiler to find the location corresponding to a name. This is essentially what we got, as the resulting car/cdr sequence corresponds to a location. What we also would expect from a compiler is to *know* that the name and name list are compile time entities, and use them as such without repeatedly testing if they are available. This is exactly what the above generated compiler fragment does *not* know, and this is why it

is so large. Another thing we can see is that unfolding strategies that rely on keeping a trace of function calls must postpone its decisions to compile time (as opposed to compiler generation time). We will later see that a strategy based on static analysis of the subject program (the interpreter) will be able to make such decisions at compiler generation time.

4.3 Binding time annotation

In an attempt to reduce the size of the generated compilers, we will make the binding times explicit in the interpreters we use for compiler generation. We assume the existence of a static analysis that can find binding times of local variables and expressions, given the information that the program given to the interpreter is static and the input is dynamic. This information will be used to annotate the interpreter: static parameters and subexpressions are marked as such and calls that should be suspended are marked as *RCALL* (for *residual call*). The annotated version of lookup is:

```
(lookup ((STATIC n) (STATIC ns) vs) =
        (IF (STATIC (EQUAL n (CAR ns)))
            (CAR vs)
            (CALL lookup n (CDR ns) (CDR vs))
)       )
```

Note that it is not necessary to annotate the actual parameters to the call as static, as the formal parameters are annotated. specialize_expression will be modified to use this information: the values of static variables are stored without a *QUOTE*, static sub-expressions are evaluated by a normal evaluation function and unfolding is decided by examining the mark.

```
specialize_expression(exp,names,values) =
   case exp of
      (STATIC e) :
         (QUOTE evaluate_expression(e,names,values))
      atomic(name) :
         lookup(name,names,values)
      (CAR e) :
         (CAR specialize_expression(e,names,values))
      (CDR e) :
         (CDR specialize_expression(e,names,values))
      (CONS e1 e2) :
         (CONS specialize_expression(e1,names,values), specialize_expression(e2,names,values))
      ⋮
      (IF (STATIC e1) e2 e3) :
         if evaluate_expression(e1,names,values) then
            specialize_expression(e2,names,values)
         else
            specialize_expression(e3,names,values)
```

```
        (IF e1 e2 e3) :
           (IF specialize_expression(e1,names,values)
               specialize_expression(e2,names,values)
               specialize_expression(e3,names,values)
           )
        (CALL f e1 ... en) :
           let (par_names,body) = definition_of(f)
           in
              let args = specialize_list(par_names,e1 ... en,names,values)
              in
                 specialize_expression(body,par_names,args)
        (RCALL f e1 ... en) :
           let (par_names,body) = definition_of(f)
           in
              let sargs = evaluate_static(par_names,e1 ... en,names,values)
                  dargs = specialize_dynamic(par_names,e1 ... en,names,values)
              in
                  (CALL (f,sargs) . dargs)


evaluate_expression(exp,names,values) =
   case exp of
      atomic(name) :
         lookup(name,names,values)
      (CAR e) :
         car(evaluate_expression(e,names,values))
      (CDR e) :
         cdr(evaluate_expression(e,names,values))
      (CONS e1 e2) :
         cons(evaluate_expression(e1,names,values), evaluate_expression(e2,names,values))
      ⋮
      (IF e1 e2 e3) :
         if evaluate_expression(e1,names,values) then
            evaluate_expression(e2,names,values)
         else
            evaluate_expression(e3,names,values)
      (CALL f e1 ... en) :
         let (par_names,body) = definition_of(f)
         in
            let args = evaluate_list(e1 ... en,names,values)
            in
               evaluate_expression(body,par_names,args)
```

specialize_list evaluates the static expressions in the list and specializes the dynamic ones. It uses the list of annotated parameter names to decide. evaluate_static evaluates the static expressions in a list and skips the dynamic, specialize_dynamic specializes the dynamic and skips the static.

evaluate_list just evaluates all expressions in the list.

Using this definition of specialize_expression yields the same result as before when specializing lookup. When we specialize specialize_expression with respect to the annotated lookup we get a quite different result:

```
specialize_expression(IF ...),(n ns vs) (values) =
    if equal(car(values),car(car(cdr(values)))) then
        (CAR car(cdr(cdr(values))))
    else
        specialize_expression(IF ...),(n ns vs)(list(car(values),
                                                     cdr(car(cdr(values))),
                                                     (CDR car(cdr(cdr(values))))
        ) )
```

which is an obviously better solution: it is much shorter, and it quite transparently builds a car/cdr sequence corresponding to the position of a variable in the name list. The only apparent inefficiency lies in the way the residual function collects all its parameters into a single list. This can be overcome *e.g.*, by post-processing the residual functions (see section 9.3).

4.4 Binding time analysis

We will now look at how we can obtain the information needed to annotate programs as in the above example. As mentioned it is *binding time information* that we need. We will require the user to specify the binding times of the program's parameters as S (for *static*) or D (for *dynamic*). Using this the *binding time analysis* will find safe approximations of the binding times of all local expressions in the program. This is then used to annotate a formal parameter as static if the actual parameter expression for that parameter is static in all calls, and as dynamic if this is not the case. Furthermore, all completely static subexpressions are marked as such, except completely static actual parameter expressions if the corresponding formal parameter is already annotated as static. In addition to these annotations, calls will be annotated as unfoldable or not unfoldable. The call annotation requires some extra analysis, and will be discussed later.

The basic idea is that an expression is dynamic if it contains a dynamic variable or calls a dynamic function. A variable is dynamic if it may be bound to the result of a dynamic expression, and a function is dynamic if its body expression is dynamic. The binding time analysis can be done as a simple abstract evaluation over a two-point domain containing the values S and D, with $S \sqsubseteq D$.

```
binding_time(exp,names,b_times) =
    case exp of
        atomic(name) :
            lookup(name,names,b_times)
        (CAR e) :
            binding_time(e,names,b_times)
        (CDR e) :
```

```
    binding_time(e,names,b_times)
(CONS e1 e2) :
    binding_time(e1,names,b_times) ⊔ binding_time(e2,names,b_times)
⋮
(IF e1 e2 e3) :
    binding_time(e1,names,b_times) ⊔
    binding_time(e2,names,b_times) ⊔
    binding_time(e3,names,b_times)
(CALL f e1 ... en) :
    binding_time_of(f)
```

The binding times of the functions and the formal parameters are held in a global environment. When handling a function call the binding time of the result is found in this environment. In addition to this, the binding times of the actual parameter expressions are found and the formal parameters are updated as the least upper bound ($\sqcup$) of the old binding time and the binding time of the actual parameter. The environment starts with all functions and all parameters bound to S, except the parameters of the goal-function, which are bound to the specified binding times. The analysis proceeds by repeatedly re-evaluating all expressions and updating the environment until it stabilizes. At this time, no identifier that has been classified as static in the environment, will at any time during partial evaluation have an unknown (non-constant) value. Using this information it is easy to annotate static formal parameters and expressions with the *STATIC* mark.

4.4.1 Call annotation

There is still the *CALL*/*RCALL* annotations to consider. The binding time information can be used to decide which calls to unfold, but some extra analysis is required. To be faithful to the binding time analysis it is necessary to unfold all calls that have a static result. Since doing this corresponds to normal evaluation it is safe (except when normal evaluation is non-terminating).

In [Sestoft 88], *inductive calls* are recognized as unfoldable. These are recursive calls where at least one static parameter is strictly decreasing, and no static parameters increase. The ordering used is a sub-structure ordering, so any value can only be decreased a finite number of times, hence it is safe to unfold the calls. The implemented method is limited to direct recursive calls, no mutual recursive inductive calls are recognized. This means that many calls that could have been safely unfolded are not. To reduce this problem, further unfolding is done in a post-process after specialization of functions.

In the *Similix* system (under development), Bondorf and Danvy use the idea that if all the conditions that control a recursion are statically decidable then the recursion can be safely unfolded at partial evaluation time. Their approach is to replace dynamic conditionals with function calls to new functions whose bodies are the replaced conditionals. These calls are not unfolded, but all others are. This means that the residual functions are not necessarily specialized versions of the original functions, but instead specialized versions of the new functions. Like Sestoft's method this approach can lead to infinite unfolding if a statically controlled recursion is non-terminating. This defect is not likely to cause problems when the partial evaluator is used for compiling or compiler generation.

Even if no infinite unfolding occur, the partial evaluation might not terminate. What can happen (as shown in section 3.2.4), is that the same function can be specialized in an infinite

number of versions. This happens if a static value increases indefinitely, usually because the condition that controls the recursion is dynamic. The proper way to handle this problem is to treat infinitely increasing static values as dynamic, modifying the result of the binding time analysis to reflect this. Recognizing precisely when a static variable increases indefinitely is equivalent to solving the halting problem, so at best we can have a good heuristic that errs on the safe side. [Jones 88] discusses this problem and suggests possible heuristics, and a current project by Jones and Andersen is implementing an algorithm involving these ideas.

In this thesis binding time analysis is used mainly as a way of recognizing which operations can be done at partial evaluation time. It is assumed that annotation of calls is done, either by hand or by an unspecified algorithm, and the problem of infinitely increasing static values is largely ignored. This is not because they are uninteresting problems, but they are outside the scope of this thesis.

Chapter 5

Revising the Mix Equation

The addition of explicit binding time arguments to the partial evaluator requires some modification of the mix equation and the Futamura projections as presented in chapter 2. This can be done by redefining *mix* to take three arguments: a program, binding time descriptions and values of static arguments or by defining it as a two-stage process: binding time analysis followed by specialization. We choose the latter alternative, as this more clearly reflect the idea of binding time analysis as a static analysis prior to specialization.

5.1 Residual programs and partial evaluators

We redefine a *residual program* of an L-program p with respect to a static input v_s to be any L-program $p^{bt}_{v_s}$ such that:

$$\mathrm{L}\ p^{bt}_{v_s}\ v_d\ =\ \mathrm{L}\ p\ combine(bt, v_s, v_d)$$

We no longer assume that specialization is with respect to the first of two arguments, so we need the binding time description bt to relate the static and dynamic values. The *combine* functions combines the static data v_s and the dynamic data v_d according to the binding time description bt. We will in this chapter only use binding time descriptions that are tuples of S's and D's, but more complex descriptions will be used in the following chapters. We will normally write the *combine* function explicitly, as it usually is trivial.

The partial evaluation is now in two steps: a binding time analyzer produces a binding time annotated p^{bt} and then a specializer produces the residual program.

$$\begin{aligned} \mathrm{L}\ bta\ (p, bt) &= p^{bt} \\ \mathrm{L}\ mix\ (p^{bt}, v_s) &= p^{bt}_{v_s} \end{aligned}$$

Combining these definitions we get the revised mix equation:

$$\mathrm{L}\ (\mathrm{L}\ mix\ ((\mathrm{L}\ bta\ (p, bt)), v_s))\ v_d\ =\ \mathrm{L}\ p\ combine(bt, v_s, v_d)$$

Note that the input language to *mix* is no longer L, but L extended with binding time annotations.

5.2 Compiling and compiler generation

Using the mix equation with p being an interpreter *s_int* and v_s being an S-program *source* we get:

$$\mathrm{L}\ (\mathrm{L}\ mix\ ((\mathrm{L}\ bta\ (s\_int, (S, D))), source))\ d\ =\ \mathrm{L}\ s\_int\ (source, d)$$

so we can obtain object programs by:

$$object\ =\ \mathrm{L}\ mix\ (s\_int^{SD}, source)$$

where

$$s\_int^{SD}\ =\ \mathrm{L}\ bta\ (s\_int, (S, D))$$

This is thus the revised first Futamura projection. The second Futamura projection requires an annotated *mix*:

$$comp\ =\ mix^{SD}_{s\_int^{SD}}\ =\ \mathrm{L}\ mix\ ((\mathrm{L}\ bta\ (mix, (S, D))), s\_int^{SD})$$

We still have

$$\mathrm{L}\ (\mathrm{L}\ comp\ source)\ d\ =\ \mathrm{S}\ source\ d$$

The compiler generator *cogen* can be generated by the third Futamura projection:

$$cogen\ =\ mix^{SD}_{mix^{SD}}\ =\ \mathrm{L}\ mix\ (mix^{SD}, mix^{SD})$$

where the annotated *mix* is obtained by binding time analysis as shown above. In the following chapters we will sometimes use

$$\mathrm{L}\ mix\ (p, v_s)$$

as a short notation for

$$\mathrm{L}\ mix\ ((\mathrm{L}\ bta\ (p, bt)), v_s)$$

when the context makes it clear what is meant.

Chapter 6

Partially Static Structures

Partially static structures cause problems when self-application of a partial evaluator is attempted. The reason is that the binding time analysis increases in complexity when its domain is non-flat. This chapter describes a method for binding time analysis, producing a context-free tree-grammar. This binding time information is then used to produce an efficient self-applicable partial evaluator. This has been realized in practice and gave satisfactory results.

First a method to do binding time analysis of partially static structures by using grammars is presented and then the actual use in a partial evaluator is detailed. It is also shown that the variable splitting that was done on the basis of user-annotation in the MIX project [Sestoft 86] can be achieved automatically and naturally by using partially static structures.

6.1 Partial evaluation with completely static / dynamic values

As seen in the preceding chapter, a simple partial evaluator will normally operate on a domain that (conceptually) is the tagged sum of values and expressions. Thus values that are known at partial evaluation time are represented by their values, while those that are unknown are represented by expressions that will evaluate to the correct values when the remaining input is supplied. These are called *residual expressions*. Examples:

$S(3)$	represents the static value 3.
$D(x+y)$	represents a dynamic value. $x+y$ is the residual expression.

In actual implementations explicit tags like S and D need not be used, since static values can be represented by constant expressions and thus be implicitly tagged. But to facilitate reading all such tags will be made explicit in this chapter.

When performing an operation the partial evaluator will test to see if the values of the parameters to the operation are static, and if they are then just perform the operation. If some were dynamic a new residual expression will be constructed from the operator and the residual expressions of the parameters.

Examples:

$$\begin{aligned} car(\ S((a\ .\ b))\ ) &= S(a) \\ car(\ D(cdr(x))\ ) &= D(car(cdr(x))) \end{aligned}$$

In the previous chapter it was argued that testing the (implicit) tags of the values made the compilers produced by self-application of the partial evaluator unsatisfactorily large and slow. To overcome this a binding time analysis was introduced. This was an abstract evaluation using a two point domain: {S,D}, where S is short for static and D for dynamic. The result of this analysis was used to annotate the program before the actual specialization. These annotations was then used instead of the tags as the basis for the tests. We will use the abbreviation "BTA" for binding time analysis.

6.2 Extending the binding time domain to partially static structures

When using structured data (like the S-expressions of LISP) it is often useful to handle values where some parts are static and other parts dynamic.

A partial evaluator handling partially static structured values in S-expression can use a domain that have partially static values in addition to the static and dynamic values of the simple partial evaluator described above. The partially static values are tagged with a P and consist of pairs of values, which again can be static, dynamic or partially static.

Examples:

$P(S(3), D(cdr(x)))$	A pair with a static head and a dynamic tail
$P(P(S(x), D(y+z)), S(nil))$	A list of (static) length 1, with a static/dynamic pair at the head

In addition to rules for partial evaluation like those before, new rules for the partially static cases are added:

$$\begin{aligned} car(P(A,B)) &= A \\ cons(S(V), D(E)) &= P(S(V), D(E)) \end{aligned}$$

A generalization of the BTA described above to such structures would operate on a domain BTA_0 with values like these:

$$P(S,D)$$
$$P(P(S,D),S)$$

for the partially static values above. These are partially ordered so D is the top element in the domain ($x \sqsubseteq D$ for all x), S is the bottom element ($S \sqsubseteq x$ for all x) and $P(x,y) \sqsubseteq P(x',y')$ iff $x \sqsubseteq x'$ and $y \sqsubseteq y'$. $P(S,S)$ is equal to S as a pair of two static values still is a static value.

The values in BTA_0 are, however, too restrictive as one (among other things) can't describe a list where the length is static (but not constant), but the values of the elements are dynamic. To do this *sets* of BTA_0 values must be used, *eg.*:

$$\{\, S\ ,\ P(D,S)\ ,\ P(D,P(D,S))\ ,\ \ldots \}$$

to describe lists of static length. The meaning of such a set, as a description of an expression, is that the expression may obtain values at partial evaluation time that at each instance will be of *one* of the forms from the set.

Since we are interested in worst case behaviour of the partial evaluator, any set containing a certain value will also contain all values that are "more static" than this. For this reason the set of *downwards closed* subsets of BTA_0 will be used instead of the powerset. This will be called BTA_1. Each element in BTA_1 is thus a downwards closed subset of BTA_0 and they will be ordered by set inclusion. A downwards closed set is a non-empty set that satisfies the following property:

if an element x is in the set, all elements y such that $y \sqsubseteq x$ will also be in the set.

For each set of BTA_0 values the *downwards closure* of that set is defined by the smallest (by set inclusion) downwards closed superset. A downwards closed set can be represented by any set, the downwards closure of which is that set. Thus the set $\{D\}$ represents the top element in BTA_1 (since its downwards closure is the whole of BTA_0). The set $\{S\}$ (which is downwards closed) is the bottom element.

Note that the set of downwards closed subsets is similar to the *Hoare power domain*, which consists of the *Scott-closed* subsets that also require that limits of chains of elements of a set also are in the set. BTA_1 is actually the Hoare power domain of the chain-completion of BTA_0. We will not need to consider limits of chains, so we use the simpler construction.

A set M seen as a BTA_1 element will (by its downwards closure) also represent all the BTA_0 values that are smaller than the elements of M by the ordering of BTA_0. Thus the set describing lists with static length and dynamic elements (shown in the example above) will, as an element of BTA_1, also represent lists where some of the elements are static (*eg.*, $P(S, P(D, S)$ and $P(D, P(S, P(D, S))))$.

Sets of BTA_0 elements can clearly be infinite (as the above example is), so a way of finitely describing such sets is needed. But as the number of elements in BTA_1 is uncountable there is no way of describing them all finitely, so we must use approximations of the elements. This can be done by constructing a subdomain BTA_2 of finitely describable elements of BTA_1, where each element in BTA_1 is represented by a finitely describable safe approximation in BTA_2. A safe approximation of an element M in BTA_1 is an element M' in BTA_2, so $M \sqsubseteq M'$ in the ordering of BTA_1 (the subset ordering).

A way of finitely describing infinite sets of tree-structures is to use *tree-grammars*, and this is what we will use here. Tree-grammars are analogous to context-free grammars and use non-terminals *etc.* in the same way. They just describe sets of tree structures instead of sets of character sequences.

$$x \rightarrow S \mid P(D, x)$$

will thus describe the set in the example above. Similarly

$$\begin{aligned} Alist &\rightarrow S \mid P(pair, Alist) \\ pair &\rightarrow P(S, D) \end{aligned}$$

will describe lists of pairs, where the first element of the pairs is static and the second is dynamic.

In our approach we will actually construct an infinite series of BTA_2 domains: BTA_{2_n}, where n is the number of different non-terminals in BTA_{2_n}. Restrictions are made on the forms of the right sides of the productions in the tree-grammar describing a BTA_{2_n}: a right side is either D, or a list of alternatives separated with |'s. The alternatives are S or $P(x, y)$ where x and y are S, D or non-terminals.

The BTA_2 value of a non-terminal or right side is the downwards closure of the set of BTA_0 values that the non-terminal (or right side) produces in the grammar. Thus D as a right side will represent the BTA_{2_n} value $\{D\}$. A non-terminal and its right side will denote the same value, so they will be used interchangeably. The requirement that non-terminals must be inside $P(,)$'s makes circular derivations impossible.

If we assume that there are no repeated alternatives on any right side and that the number of different non-terminals used on right sides is a finite number, n, we will have a finite number of possible right sides, and thus a finite BTA_{2_n} domain. However, one may note that two different right sides can very well represent the same BTA_{2_n} value. This is not important in the present application, and it is not difficult to see when two right sides represent the same value, an algorithm deciding equivalence is fairly easy to write.

The least upper bound of two BTA_{2_n} values in a certain grammar will be the smallest BTA_{2_n} value that safely approximates the least upper bound of the values seen as elements in BTA_1, which is the (downwards closure of) the union of the sets of BTA_0 values that represents the BTA_{2_n} values.

A right side representing this can be constructed by adding the alternatives of the right sides representing the two values, unless one of the right sides is D, in which case the least upper bound is just D, as this is the top element.

In binding type analysis the non-terminals will correspond to variables or functions in a program, and the values of them will describe the values that the variable or function can obtain during partial evaluation.

The analysis will start by assigning the value S to all non-terminals except those that represent the parameters to the goal function, and then do a fixed point iteration where at each step the non-terminals will be updated with new values. When a non-terminal changes value all the right sides that refer to it will also change values *etc.*, but as all changes are monotonic in the ordering of BTA_{2_n}, the result will still be a safe approximation, only it is less precise.

Note that the abstract evaluation handles values in the form of right sides to a grammar. The actual values these right sides represents are the downwards closures of the sets they derive relative to the grammar.

A tree grammar is equivalent to a set of recursive set-equations, and methods like the ones used in [Reynolds 69] or [Jones 86] could be used to obtain such descriptions. The method shown below is, however, simpler in the treatment of selectors (eg. *car* and *cdr*).

Part of the difficulty of the BTA stems from the fact that the language is untyped: the analysis must do a kind of type inference to find good binding time information. With a typed language a fixed set of BT values can be found for each type, making the use of grammars unnecessary. The projections in [Launchbury 88] use this idea, which is further investigated in chapter 8.

6.3 Binding time analysis

To make a partial evaluator self-applicable, one must be able to find BTA values automatically. In the case of the two-value domain this can be done with a simple abstract evaluation over a two-value domain, but when recursive descriptions like grammars are required, it is not obvious how it can be done.

The language that is used is called L and is a small functional language similar to that in section 3.1 with a few additions. Figure 6.3 presents the syntax of the language.

Figure 6.3. The syntax of the language L.

$\langle program \rangle \rightarrow (\langle function \rangle^*)$

$\langle function \rangle \rightarrow (\langle name \rangle\ (\langle name \rangle^*)\ \langle exp \rangle)$

$\langle exp \rangle \rightarrow \langle name \rangle$
$\mid (quote\ \langle constant \rangle)$
$\mid (car\ \langle exp \rangle)$
$\mid (cdr\ \langle exp \rangle)$
$\mid (atom\ \langle exp \rangle)$
$\mid (null\ \langle exp \rangle)$
$\mid (cons\ \langle exp \rangle\ \langle exp \rangle)$
$\mid (equal\ \langle exp \rangle\ \langle exp \rangle)$
$\mid (if\ \langle exp \rangle\ \langle exp \rangle\ \langle exp \rangle)$
$\mid (let\ (\langle binding \rangle^*)\ \langle exp \rangle)$
$\mid (call\ \langle name \rangle\ \langle exp \rangle^*)$

$\langle binding \rangle \rightarrow \langle name \rangle = \langle exp \rangle$

The semantics are straight-forward. The program is a list of function definitions where the first is the goal function. The scope of each function is the whole program, and there are no local function definitions. The calling mechanism is applicative (call-by-value). The parameters and other variables are statically scoped.

An expression is either a name, which represents a variable, or a list consisting of an operator followed by one or more arguments. The operators *quote*, *car*, *cdr*, *atom*, *null*, *cons* and *equal* behave exactly as in LISP. *if* is an if-then-else operator with the usual semantics. *let* defines a set of new name/value bindings with a local scope (as in ML or Scheme). *call* is the function application operator. All parameters to a function must be supplied when it is called, so there will be no building of closures and thus no higher order functions.

The idea is to use names from the program as non-terminals in the BTA descriptions, so the non-terminal corresponding to a variable would describe the set of values that the variable can take, and non-terminals for function names describe the set of possible result values of the function.

Since the right sides of the productions can refer to other non-terminals, this approach requires all variables to have distinct names even if they have disjoint scopes (actually this is not strictly required, as the BTA would still give a safe result: it would only be less precise). Due to the limitations to the right sides involving $P(x, y)$ there are further restrictions. Since the x's and y's can either be S, D or non-terminals, it would help if the parameter expressions of *cons* were of forms that are always describable by such values. Such forms are constants (describable by S) or variables and function calls (describable by non-terminals). This property is easily achieved by adding *let*-expressions where necessary.

There are thus some small changes in the syntax of the language that is required as input to the BTA (figure 6.3).

The BTA is computed by a fixed-point iteration that starts with a global grammar containing

Figure 6.3. The changes in syntax for the input to the BTA.

⟨*exp*⟩	—	⋮
	|	(*cons* ⟨*restricted-exp*⟩ ⟨*restricted-exp*⟩)
		⋮
⟨*restricted-exp*⟩	—	⟨*name*⟩
	|	(*quote* ⟨*constant*⟩)
	|	(*call* ⟨*name*⟩ ⟨*exp*⟩*)

S productions for all non-terminals except those describing the values of the parameters to the the goal-function. These are given productions describing their actual binding times (usually S or D). Alternatives are added to the productions until a fixed-point is achieved. As the number of non-terminals is a fixed number n, the number of right sides is limited. Since all changes are monotonic in the domain of BTA_{2_n} values, there will be a finite fixed-point grammar which can be obtained in a finite number of steps. Such a fixed-point iteration is well known from flow analysis (eg. [Mycroft 80]).

Each step in the iteration recomputes the values of expressions each time the variables or functions they use change value. This is done by evaluating the expressions in the BTA domain. Instead of checking for each expression if it needs to be recomputed, *all* expressions are recomputed at each iteration step until no changes occur between two such steps.

The abstract evaluation is described in section 6.8 at the end of this chapter. The values that are actually computed are right sides of productions, but when alternatives are added to a right side in the global grammar, this changes the meaning of a non-terminal, and thus the right sides that use it. The places in the abstract evaluation where the updating occurs are marked by comments. The updating is done by replacing a right side with the least upper bound of the old value and the new.

As an example, here is a function which constructs an a-list structured environment from a list of names and a list of values.

```
(Make_env (names values)
          (if (null names)
              (quote nil)
              (let
                  (hd = (let (hd1 = (car names) tl1 = (car values)) (cons hd1 tl1)))
                  (cons hd (call Make_env (cdr names) (cdr values)))
)         ) )
```

Assume that the function Make_env has static first parameter (*names* — S) and dynamic second parameter (*values* — D), then the initial environment would be:

names	— S
values	— D
*hd*1	— S
*tl*1	— S
hd	— S
Make_env	— S

The first iteration step of the binding time analysis would yield this environment:

$$\begin{array}{ll} names & \rightarrow S \\ values & \rightarrow D \\ hd1 & \rightarrow S \\ tl1 & \rightarrow D \\ hd & \rightarrow S \mid P(S,D) \\ Make\_env & \rightarrow S \mid P(hd,S) \end{array}$$

After the second iteration step the value of Make_env is $S \mid P(hd,S) \mid P(hd,Make\_env)$ which is unchanged after the third step, giving this final environment:

$$\begin{array}{ll} names & \rightarrow S \\ values & \rightarrow D \\ hd1 & \rightarrow S \\ tl1 & \rightarrow D \\ hd & \rightarrow S \mid P(S,D) \\ Make\_env & \rightarrow S \mid P(hd,S) \mid P(hd,Make\_env) \end{array}$$

The $P(hd,S)$ alternative in the value of Make_env is redundant since all the values could be obtained from the other two alternatives alone. Such redundancies often occur, but since they do not influence the later use of the description no attempt is made to reduce the environment.

The number of iterations required before a fixed-point is reached depends on the program that is analyzed, but with the optimizations that are included in the actual implementation of the BTA most programs need only four iterations.

6.4 Annotation

The binding time information is used for annotation of the program in question. Expressions that operate entirely on static data are marked by putting the tag S in front of them. Other expressions are marked operator by operator. The operators that have some possibly dynamic (D) operands are marked by adding a D to the operator name, so *car* becomes *carD etc.* The remaining operators are partially (or possibly completely) static and they are marked with a P.

There are some special cases with *cons*, *if*-expressions, *let*-expressions and function calls. *cons* is not marked as it doesn't have to access the operands (in terms of partial evaluation it is non-strict in both arguments). *if*-expressions are only marked after the condition (they are strict only in their first argument). In *let*-expressions each binding is marked if it can be statically determined that the variable will occur at most once in any residual program, in which case it can be safely unfolded at partial evaluation time. If all bindings can be unfolded the let-expression is marked as eliminable by adding an E to the operator. It can be determined from the BTA-values of the formal parameters of a function, whether the actual parameters are static *etc.*, so no extra marking is used there. Function calls are marked to indicate whether they should be unfolded (β-reduction) or left as calls in the residual program. Residual calls are marked by changing the *call* operator to *rcall*. At present this is done by hand, but methods like those described at the end of the revious chapter can be used.

Several other annotations are used. In general as many as possible of the tests that the partial evaluator would otherwise make at partial evaluation (dynamic) time are determined by using the binding time information (at static time). The results of these tests are added to the program as annotation of operators *etc.*

The example program from above would yield the following annotated program:

```
(Make_env (names values)
          (ifS (null names)
               (S (quote nil))
               (letE
                    (hd =1 (letE (hd1 =1 (S (car names)) tl1 =1 (carD values))
                                 (cons (S hd1) tl1)
               )    )
               (cons hd (call Make_env (S (cdr names)) (cdrD values)))
)          )  )
```

6.5 Function specialization

The previous phases (BTA and annotation) are done at *metastatic* time, that is only knowing *which* of the parameters of the subject program (the program that is being partially evaluated) are static. When the *values* of these parameters become known (at *static* time) a residual program is constructed by making specialized versions of the functions in the subject program. The residual program can then be given various values of the dynamic parameters (at *dynamic* time) to produce final results.

The functions are specialized with respect to the static parts of their parameters. The parameters to a specialized function correspond to the dynamic parts of the parameters to the original function. The static parts are absorbed into the residual expression that becomes the body of the new function.

The static parts of the parameters to the Make_env function above are the contents of the *names* variable, and the dynamic parts are the contents of the *values* variable. Assume the value of *names* is $(x\ y\ z)$, then the specialized version would be:

```
(Make_env1 (values)
           (cons (cons 'x (car values))
                 (cons (cons 'y (cadr values))
                       (cons (cons 'z (caddr values))
                             'nil
)          )     )     )
```

where $'x$, (*cadr values*) *etc.* are the usual shorthand for (*quote* x), (*car* (*cdr values*)) *etc.*

Note that the recursive calls have been unfolded and that the structure of the static variables has been absorbed into the structure of the new function body. Often the value will be completely absorbed, so no part of it can be directly seen in the new body (unlike $'x$ *etc.*).

If a function call is not unfolded the residual expression will contain a call to a specialized version of the called function. Specialized functions can be uniquely identified by the original function name and the values of the static parts of the parameters. The specialized function above (Make_env_1) is thus identified by the name Make_env and the static value $(x\ y\ z)$.

The body of a specialized function is of course an expression, but the result of partially evaluating the original expression is a value in the PE domain (using S D and P), so this value is made into an expression by making the static parts into constant expressions, and making *cons* expressions for P terms. In the Make_env example above the value

```
P( P(S(x),D((car values))),
   P( P(S(y),D((cadr values))),
      P( P(S(z),D((caddr values))), S(nil) )
) )
```

was converted to the expression shown as the body to the residual function in the example above. If the value was used as a parameter to a residual (not unfolded) call to a function f, the static parts are used to identify the specialized function and the dynamic parts are used as parameters to the function. This would give the residual expression

```
(call f1 (car values) (cadr values) (caddr values))
```

where f_1 is identified by the original name (f) and the values of the static parts:

```
P( P(S(x),D(_)),
   P( P(S(y),D(_)),
      P( P(S(z),D(_)), S(nil) )
) )
```

As can be seen from the example the static parts are obtained by blanking out the dynamic parts. The static parts thus reflects the structure of the value and it can be used to see how many dynamic parts there are. This is important because the number of dynamic parts is needed to find the number of parameters to the specialized function. As the example shows, a single parameter in the original function call can result in several parameters in the residual call, so the definition of the specialized version must reflect this by having the same number of formal parameters. This automatically realizes the *variable splitting* that was done in [Sestoft 86] on the basis of an annotation by hand, and is an alternative to the retyping described in section 9.3.

During the partial evaluation the annotations in the program are used. The expressions that are marked as *S*tatic are thus just evaluated without testing whether the variables in them have static values (they will have, the BTA ensures that). The operators marked *D*ynamic are used to construct new expressions even if the arguments by chance are static (remember that *D* means *possibly* dynamic). This makes the partial evaluator more conservative than a naive partial evaluator (without BTA), but experience has shown that this rarely causes problems and that it in some cases improves the quality of residual programs by avoiding generation of almost identical specialized functions. This can *e.g.*, happen when a function uses an accumulating parameter to build the result (like in tail-recursive list-reversal). This parameter will initially be the constant *nil* which is a static value but later recursions will change this to a dynamic value. The naive partial evaluator will generate a specialized version for the *nil* case and another for the general (dynamic) case, whereas the BTA will show that the value is actually dynamic and thus only generate a specialized function for the general case. Note that with the naive partial evaluator no residual program can contain function calls with constant parameters as the static parts of the parameters are absorbed. It is generally a good idea to avoid such restrictions on residual programs.

During the function specialization phase lists are kept of descriptions of specialized functions that have been generated and of specialized functions that have been called but not yet generated. The latter list initially contains a description consisting of the name of the goal-function and the values of the static parameters to the program. When it becomes empty all the needed specialized functions are generated and the partial evaluation is finished. There are cases when

this will not happen: either a recursive function call is unfolded infinitely, or an infinite number of specialized functions are generated. The first case can be solved by marking the call as *not* unfoldable, but this will in some cases just change the problem to the other type. Infinitely many specialized functions occur when the static parts of the parameters to a function obtain infinitely many different values during the function specialization phase. This is more difficult to solve. It is possible to restrict the number of possible values by forcing some of the static parts of a variable to be dynamic by giving them dynamic initial descriptions for the BTA (so it becomes not only the parameters to the goal-function that should be specified). This will however in some cases give too conservative descriptions for other variables (as a consequence of the BTA). Another solution is to restructure the program so the problem doesn't occur. This can be *very* tricky, and in a few cases one must conclude that for some problems it is not possible (in the present framework) to find programs where the partial evaluation will terminate with a non-trivial solution.

An extended example showing compilation by partially evaluating an interpreter is shown in sections 6.9 to 6.11. Timings are shown in the next section.

6.6 Results

The partial evaluator has been implemented and satisfactory results have been obtained. Figure 6.6 shows some execution times. The system is implemented using Franz Lisp on a Vax 785. All executions are done using compiled LISP programs. In addition to the total execution time the table shows how much of that is spent doing garbage collection (GC). Where applicable a ratio of execution times for original and residual programs is shown.

The program *Btann* performs both BTA and annotation, of which the BTA is the major part. *Fsp* performs function specialization. The notation $L\ program\ \langle x_1, \ldots, x_n \rangle$ represents execution of the L program *program* with input $x_1, \ldots, x_n$. $program_{ann}$ represents the annotated version of *program*.

Two interpreters are used for the executions. *mp_interp* is the interpreter from section 6.9. *self_interp* is an interpreter for L. Since the source and object languages for compilations using a self-interpreter is the same, they should ideally be near-identity mappings or at least not yield programs that are markedly less efficient than the originals. For this system such compilations are in fact near-identity mappings. Apart from renaming and permutation of functions and variables and the addition of an extra goal function (that splits the *list* of parameters that the interpreter requires into *separate* parameters when calling the new version of the original goal function) there are virtually no differences between the original program and the new. In addition to comparing execution times for *self_interp* interpreting a L-program and the compiled L-program the table also compares the compiled program with the original program (this makes sense as they are in the same language).

The results from this table compares satisfactorily to the results in [Sestoft 86]. Though the time used to partially evaluate programs is longer, especially for the BTA phase, the compiled programs are faster. The ratio between compiling with the partial evaluator and a generated compiler is lower than in [Sestoft 86] mainly because the present partial evaluator does more manipulation on dynamic data (such manipulations are not improved by knowledge of static data). On the other hand the ratio between interpreting a program and executing a compiled program is higher, which is partly due to the extra manipulation of dynamic data and the automatic splitting of parameters.

Figure 6.6 shows the sizes of the programs. Three numbers are shown: the number of tokens,

Figure 6.6 Execution times for the partial evaluator and residual programs.			
job	exec. time	GC time	ratio
$Fsp_{ann} = L\ Btann\ \langle\ Fsp\ \rangle$	196160 ms	56560 ms	
$cogen = L\ Fsp\ \langle\ Fsp_{ann}, Fsp_{ann}\ \rangle$	352460 ms	73980 ms	
$cogen = L\ cogen\ \langle\ Fsp_{ann}\ \rangle$	88360 ms	50620 ms	3.9
$mp\_interp_{ann} = L\ Btann\ \langle\ mp\_interp\ \rangle$	6560 ms	620 ms	
$l\_expo = L\ Fsp\ \langle\ mp\_interp_{ann}, mp\_expo\ \rangle$	2560 ms	540 ms	
$l\_expo = L\ mp\_compiler\ \langle\ mp\_expo\ \rangle$	360 ms	0 ms	7.1
$output_1 = L\ mp\_interp\ \langle\ mp\_expo, input_1\ \rangle$	7520 ms	0 ms	
$output_1 = L\ l\_expo\ \langle\ input_1\ \rangle$	200 ms	0 ms	37
$mp\_compiler = L\ Fsp\ \langle\ Fsp_{ann}, mp\_interp_{ann}\ \rangle$	35760 ms	8720 ms	
$mp\_compiler = L\ cogen\ \langle\ mp\_interp_{ann}\ \rangle$	5520 ms	2540 ms	6.4
$self\_interp_{ann} = L\ Btann\ \langle\ self\_interp\ \rangle$	5100 ms	560 ms	
$self\_int' = L\ Fsp\ \langle\ self\_interp_{ann}, self\_interp\ \rangle$	25920 ms	10560 ms	
$self\_int' = L\ self\_compiler\ \langle\ self\_interp\ \rangle$	2560 ms	500 ms	10
$output_2 = L\ self\_interp\ \langle\ self\_interp, input_2\ \rangle$	48940 ms	2400 ms	
$output_2 = L\ self\_int'\ \langle\ input_2\ \rangle$	1100 ms	0 ms	44
$output_2 = L\ self\_interp\ \langle\ input_2\ \rangle$	1080 ms	0 ms	1.02
$self\_compiler = L\ Fsp\ \langle\ Fsp_{ann}, self\_interp_{ann}\ \rangle$	41700 ms	11600 ms	
$self\_compiler = L\ cogen\ \langle\ self\_interp_{ann}\ \rangle$	5580 ms	2440 ms	7.4

the number of functions and a ratio. The number of tokens is measured by adding the number of CONS-cells to the number of non-NIL atoms in the data structures. L-programs are measured *before* the restrictions from fig. 3.2 are applied, as this gives the fairest comparison with residual programs where these restrictions do not apply. The number of functions only apply to L-programs. The ratio is between the number of tokens in an interpreter and the corresponding compiler.

The compiler/interpreter ratio for the small interpreters are fairly large compared to the ratio between *cogen* and *Fsp*. This is because the compilers consist of a constant part which is virtually the same in all compilers and a interpreter-dependent part which is roughly linear in the size of the interpreters, so the ratio converges to about 3.3 when the size of the interpreter increases.

The names of the programs are the same as in figure 6.6.

Figure 6.6 Sizes of various programs and residual programs.			
program	tokens	functions	ratio of tokens
Fsp	13172	66	
cogen	44545	105	3.4
mp_interp	903	8	
mp_compiler	11282	51	12.5
mp_expo	171		
l_expo	293	4	
self_interp	985	8	
self_compiler	11830	54	12.0
self_int'	1009	9	

6.7 Conclusion

The main purpose of handling partially static structures in a partial evaluator is to increase the generality of it, that is extend the class of programs that are handled non-trivially. Experience shows that programs using structures that will be partially static/dynamic at partial evaluation time normally can be rewritten so all variables/functions are either totally static or totally dynamic. This however puts extra demands on the user and often decreases the efficiency of the program.

In the sense of extending the class of programs that are handled non-trivially by the partial evaluator the project has been a success. Even so, it is still sometimes necessary to rewrite programs to get good results from the partial evaluator.

In addition to this generalization the method gives a natural and automatic way of achieving the variable-splitting that in [Sestoft 86] is done in an *ad hoc* fashion on the basis of user annotation. Another way of achieving variable-splitting automatically is described in section 9.3.

The quality of residual programs is very good, in the sense that there is little that obviously can be done to improve the programs obtained by partial evaluation of the MP interpreter and the self-interpreter. The compilers are fairly large and slow compared to those achieved by later versions of the partial evaluators described in [Jones *et al.* 85] and [Sestoft 86], but this is mostly due to greater complexity of the partial evaluator of which they are residual programs.

As mentioned earlier the call unfolding are still made according to hand-made annotations as in [Jones *et al.* 85]. Something similar to the combined pre- and post-processing of [Sestoft 88] could be used, though it would require an extension of the static analysis in the pre-process to handle calls to functions with partially static results.

The handling of the partially static structures has given rise to some subtle problems in connection with some language constructions. As an example imagine a let-expression of the form:

(let (bindings to dynamic variables) partially static expression)

The bindings are needed to define variables in the expressions that form the dynamic parts of the value of the partially static expression, and should thus be a part of the result of the let-expression. But the result of the partially static body must be of a form using $P(x, y)$, where both x and y can contain dynamic parts, so the question is where to put the bindings. A possible way to do this is to copy the let-bindings onto each of the expressions in the partially static value. This would, however, cause duplication of code and possibly repeated calculations in the residual programs. Another possibility would be to make the value of the let-expression completely dynamic so there would be only one expression to add the bindings to. This is however unsatisfactory due to the loss of static data. The solution that is used is keeping a field in the representation of partially static structures that contain bindings that are *common* to all expressions in the structure. These bindings are copied along when access is made to substructures, and when two structures are concatenated a union of the bindings are made. This requires some rather complex manipulations to ensure that the bindings are kept in the proper order, but it solves the problem satisfactorily.

The way tree grammars are used in the BTA can be extended to a framework that is useful for other kinds of abstract interpretation, such as type inference and strictness analysis.

The way BTA is handled in this chapter is an operational way; it approximates sets of values in a naive partial evaluator. But a naive partial evaluator, and thus also a partial evaluator using an operational BTA, can build infinite sets of static structures because the conditions that limit

the sets are dynamic. Such values should be regarded as dynamic rather than static, making the term *dynamic* apply to all values that are not limited at static time. Such a *semantic* (as opposed to operational) BTA that is sufficiently liberal (not making too much dynamic) has not yet been achieved, but work is being done in Copenhagen towards this goal.

6.8 Abstract evaluation in the BTA domain.

Only the abstract interpretation of expressions is shown. The values are represented by right sides of a grammar. The actual values are the downwards closures of the sets the right sides derive relative to the grammar.

$Aeval(name)$ = the right side of *name* in the global grammar

$Aeval((quote\ constant)) = S$

$Aeval((car\ exp)) = Car^*(Aeval(exp))$

$Aeval((cdr\ exp)) = Cdr^*(Aeval(exp))$

$Aeval((atom\ exp)) = Aeval((null\ exp)) =$
$\quad let\ val = Aeval(exp)\ in$
$\quad\quad if\ val = D\ then\ D$
$\quad\quad else\ S$

$Aeval((cons\ rexp1\ rexp2)) =$
$\quad let\ rval1 = Reval(rexp1)\ ,\ rval2 = Reval(rexp2)\ in$
$\quad\quad if\ rval1 = S\ and\ rval2 = S\ then\ S$
$\quad\quad else\ P(rval1,\ rval2)$

$Aeval((equal\ exp1\ exp2)) =$
$\quad let\ val1 = Aeval(exp1)\ ,\ val2 = Aeval(exp2)\ in$
$\quad\quad if\ val1 = S\ and\ val2 = S\ then\ S$
$\quad\quad\quad (*\ val1 = S$ means that there is only one $*)$
$\quad\quad\quad (*$ alternative on the right side, and that is $S\ *)$
$\quad\quad else\ D$

$Aeval((if\ exp1\ exp2\ exp3)) =$
$\quad let\ val1 = Aeval(exp1)\ ,$
$\quad\quad val2 = Aeval(exp2)\ ,$
$\quad\quad val3 = Aeval(exp3)\ in$
$\quad\quad else\ if\ val1 = D\ then\ D$
$\quad\quad else\ Least\_upper\_bound(val2, val3)$

$Aeval((let\ bindings\ exp)) = Aeval(exp)$
$\quad\quad (*$ Here the non-terminals in the global grammar corresponding to the names of $*)$
$\quad\quad (*$ the bindings are updated by the abstract values of the corresponding expressions. $*)$

Aeval((*call name . actual_params*)) = the right side of *name* in the global grammar
(* Here the non-terminals in the global grammar corresponding to the names *)
(* of the formal parameters of the function are updated by the *)
(* abstract values of the actual parameters. *)
(* *Reval* evaluates restricted expressions returning restricted values. *)

Reval(*name*) =
let val = the right side of *name* in the global grammar *in*
if val = *S or val* = *D then val*
else name (* non-terminal *)

Reval((*quote constant*)) = *S*

Reval((*call name . actual_params*)) =
let val = the right side of *name* in the global grammar *in*
if val = *S or val* = *D then val*
else name (* non-terminal *)
(* Here the non-terminals in the global grammar corresponding to the names *)
(* of the formal parameters of the function are updated by the *)
(* abstract values of the actual parameters. *)

(* Car^* and Cdr^* evaluates head and tail of abstract values *)

$Car^*(D) = D$
$Car^*(S) = S$
$Car^*(P(x,y)) =$
if $x = S$ *or* $x = D$ *then* x
else the right side of x in the global grammar
$Car^*(a \mid b) = Least\_upper\_bound(Car^*(a), Car^*(b))$
$Cdr^*(D) = D$
$Cdr^*(S) = S$
$Cdr^*(P(x,y)) =$
if $y = S$ *or* $y = D$ *then* y
else the right side of y in the global grammar
$Cdr^*(a \mid b) = Least\_upper\_bound(Cdr^*(a), Cdr^*(b))$

(* *Least_upper_bound*evaluates the least upper bound of two abstract values *)

$Least\_upper\_bound(D,x) = D$
$Least\_upper\_bound(x,D) = D$
$Least\_upper\_bound(S,S) = S$
$Least\_upper\_bound(x,y) = x \mid y$
(* The alternatives from both right sides with duplicate alternatives removed *)

6.9 An interpreter for MP

The interpreter is shown in the syntax that is required by the partial evaluator. It was originally written using a syntactically sugared form which was translated into what is shown here. *xcall* calls an "external" function which is not defined in the program but in the run-time environment.

The language MP is taken from [Sestoft 86]. It is a simple imperative language using while-statements for repetition. The interpreter differs from the one in [Sestoft 86] in using an environment that is a list of name/value pairs instead of using separate name and value lists.

Each MP program declares a list of input variables and a list of other variables. The input variables are initialized with variables from the input list and the others with the value *nil*. The result of interpreting a MP program is the final values of all variables represented by the final environment.

```
(
  (Mp (program input)
      (let (l4 = (car (cdr program))
            l5 = (car (cdr (cdr program)))
            )
      (rcall Block (car (cdr (cdr (cdr program))))
                   (call Initvars (cdr l4) (cdr l5) input)
) )    )

(Initvars (parlist varlist input4)
          (if (null parlist)
              (if (null varlist) 'nil
                  (let (hd9 = (let (hd8 = (car varlist)) (cons hd8 input4)))
                       (cons hd9 (call Initvars parlist (cdr varlist) input4))
              ) )
              (let (hd7 = (let (hd5 = (car parlist) tl6 = (car input4)) (cons hd5 tl6)))
                   (cons hd7 (call Initvars (cdr parlist) varlist (cdr input4)))
)         ) )

(Block (block env)
       (let (l6 = (car block))
            (if (null block) env
                (if (equal (car l6) ':=)
                    (call Block (cdr block)
                                (call Update env
                                              (car (cdr l6))
                                              (call Exp (car (cdr (cdr l6))) env)
                    )           )
                (if (equal (car l6) 'if)
                    (if (call Exp (car (cdr l6)) env)
                        (call Block (call Append (car (cdr (cdr l6))) (cdr block)) env)
                        (call Block (call Append (car (cdr (cdr (cdr l6)))) (cdr block)) env)
                    )
                (if (equal (car l6) 'while) (rcall While block env)
                    (xcall writeln 'Unknown_command (car block))
```

```
    )   )  ))))

    (While (block11 env10)
           (if (call Exp (car (cdr (car block11))) env10)
              (call Block (call Append (car (cdr (cdr (car block11)))) block11) env10)
              (call Block (cdr block11) env10)
    )      )

    (Exp (exp env12)
         (if (atom exp) (call Lookup exp env12)
         (if (equal (car exp) 'quote) (car (cdr exp))
         (if (equal (car exp) 'car) (car (call Exp (car (cdr exp)) env12))
         (if (equal (car exp) 'cdr) (cdr (call Exp (car (cdr exp)) env12))
         (if (equal (car exp) 'atom) (atom (call Exp (car (cdr exp)) env12))
         (if (equal (car exp) 'cons)
            (cons (call Exp (car (cdr exp)) env12) (call Exp (car (cdr (cdr exp))) env12))
         (if (equal (car exp) 'equal)
            (equal (call Exp (car (cdr exp)) env12) (call Exp (car (cdr (cdr exp))) env12))
         (xcall writeln 'Unknown_expression exp)
    )    )))))))

    (Update (env13 var val)
            (if (null env13) (xcall writeln 'Unknown_variable var)
               (if (equal (car (car env13)) var)
                  (let (hd15 = (cons var val) tl16 = (cdr env13)) (cons hd15 tl16))
                  (let (hd14 = (car env13)) (cons hd14 (call Update (cdr env13) var val)))
    )       )  )

    (Lookup (var18 env17)
            (if (null env17) (xcall writeln 'Unknown_variable var18)
               (if (equal (car (car env17)) var18)
                  (cdr (car env17))
                  (call Lookup var18 (cdr env17))
    )       )  )

    (Append (a b)
            (if (null a) b (let (hd19 = (car a)) (cons hd19 (call Append (cdr a) b))))
    )
)
```

6.10 Exponentiation program in MP.

This is an exponentiation program in MP. It is taken from [Sestoft 86].

Input: Two lists x and y.

Output: The variable out is a list the length of which is the number of all tuples of length $|y|$ of elements from x, which is $|x| \uparrow |y|$.

```
(program (pars x y) (dec out next kn)
        ((:= kn y)
        (while kn
              ((:= next (cons x next))
              (:= kn (cdr kn))
              )
        )
        (:= out (cons next out))                                  ; First combination
                                                                  ; Invariant: |next| + |kn| = |y|
        (while next                                               ; while more tuples
              ((if (cdr (car next))                               ; if next(1) can be increased
                  ((:= next (cons (cdr (car next)) (cdr next)))   ; do that
                  (while kn                                       ; while |next| < |y| do
                        ((:= next (cons x next))                  ; put x in front of next
                        (:= kn (cdr kn))                          ; preserving invariant
                        )
                  )
                  (:= out (cons next out))
                  )
                                                                  ; else, backtrack, preserving invarian
                  ((:= next (cdr next))
                  (:= kn (cons '1 kn))
                  )
              ))
        )
        )
)
```

6.11 Residual exponentiation program.

This program is the residual program obtained by partially evaluating the MP interpreter from section 6.9 with respect to the exponentiation program from section 6.10. It accepts a list of input parameters (of length one) and returns a final binding of variables to values. Note that the environment from the interpreter is split into several parameters to each function (all the names beginning with *env*), each of which represent a variable from the MP program.

```
((Mp (input-4)
     (call Block-5 (car input-4)
                   (car (cdr input-4))
                   (cdr (cdr input-4))
                   (cdr (cdr input-4))
                   (cdr (cdr input-4))
)    )

(Block-5 (env-10 env-9 env-8 env-7 env-6)
        (call While-11 env-10 env-9 env-8 env-7 env-9)
```

```
)

(While-11 (env10-16 env10-15 env10-14 env10-13 env10-12)
          (if env10-12
             (call While-11 env10-16 env10-15 env10-14 (cons env10-16 env10-13) (cdr env10-12))
             (call While-19 env10-16 env10-15 (cons env10-13 env10-14) env10-13 env10-12)
)         )

(While-19 (env10-24 env10-23 env10-22 env10-21 env10-20)
          (if env10-21
             (if (cdr (car env10-21))
                (call While-11 env10-24
                               env10-23
                               env10-22
                               (cons (cdr (car env10-21)) (cdr env10-21))
                               env10-20
                )
                (call While-19 env10-24 env10-23 env10-22 (cdr env10-21) (cons '1 env10-20))
             )
             (cons (cons 'x env10-24)
                   (cons (cons 'y env10-23)
                         (cons (cons 'out env10-22)
                               (cons (cons 'next env10-21)
                                     (cons (cons 'kn env10-20) 'nil)
))        ) )     )     )     )
```

Chapter 7

Separating Binding Times

In chapter 6, we presented a binding time analysis and a partial evaluator capable of handling partially static structures. The partial evaluator was, however, somewhat larger than a simple partial evaluator capable only of handling completely static / dynamic, and the generated compilers also became larger and slightly slower, as these were specializations of a larger partial evaluator.

This chapter presents a method for converting a program containing variables of mixed binding times into a program with strong separation of binding times. It starts by using an advanced binding time analysis to classify partially static variables, and then using this information to do an automatic staging transformation [Jørring *et al.* 86] of the program, to separate binding times. After this separation, all variables and functions are either completely static or completely dynamic, and the program can thus be partially evaluated using the simple partial evaluator.

The intuition is that each partially static variable is split into two: one holding the static part and another holding the dynamic part, and each function returning a partially static result is split into two: one returning the static part of the result given the static part of the parameters, and another returning the dynamic part of the result given both the static and the dynamic parts of the parameters. The function that returns the dynamic part will need both, as some run time values can depend on compile time values (whereas compile time values shouldn't depend on run time values).

The resulting program will be equivalent to the original, but there are now only completely static and dynamic variables and functions.

As an example of what we expect as a result of the transformation, consider the function below:

$$
\begin{array}{l}
\mathit{Make\text{-}A\text{-}list}(\mathit{names}, \mathit{values}) \; = \\
\quad \textbf{if } \mathit{atom}(\mathit{names}) \textbf{ then } \mathit{'nil} \\
\quad \textbf{else } ((\mathit{car}(\mathit{names}) \; :: \; \mathit{car}(\mathit{values})) \\
\qquad\qquad :: \; \mathit{Make\text{-}A\text{-}list}(\mathit{cdr}(\mathit{names}), \mathit{cdr}(\mathit{values})))
\end{array}
$$

*Make-A-list*takes a list of names and a list of values and returns a list of name/value pairs. Assuming names is static and values is dynamic, we can transform this function into the following two new functions, returning respectively the static and the dynamic part of the result:

$$
\begin{array}{l}
\mathit{Make\text{-}A\text{-}list}_s(\mathit{names}) = \\
\quad \textbf{if } \mathit{atom}(\mathit{names}) \textbf{ then } '\mathit{nil} \\
\quad \textbf{else } ((\mathit{car}(\mathit{names}) :: '\bot) \\
\qquad\qquad :: \mathit{Make\text{-}A\text{-}list}_s(\mathit{cdr}(\mathit{names})))
\end{array}
$$

$$
\begin{array}{l}
\mathit{Make\text{-}A\text{-}list}_d(\mathit{names}, \mathit{values}) = \\
\quad \textbf{if } \mathit{atom}(\mathit{names}) \textbf{ then } '\bot \\
\quad \textbf{else } (('\bot :: \mathit{car}(\mathit{values})) \\
\qquad\qquad :: \mathit{Make\text{-}A\text{-}list}_d(\mathit{cdr}(\mathit{names}), \mathit{cdr}(\mathit{values})))
\end{array}
$$

$\mathit{Make\text{-}A\text{-}list}_s$, when given the static parameter (names) returns the static part of the result and $\mathit{Make\text{-}A\text{-}list}_d$ returns the dynamic part given both arguments. The parts that are not necessary are replaced by $\bot$, which in this context means "not needed", and can be represented by an arbitrary value. It will not mean "nonterminating". Note that only the completely static or dynamic parts are replaced, the common structure is retained in both functions.

Plan

Section 7.1 describes how the grammar is obtained given the binding times of the input to the program.

The binding time analysis used is a variant of the one from chapter 6, using regular tree grammars to describe mixed binding times. Each identifier in the program is assigned a nonterminal in the grammer, which will produce the set of binding time patterns of that identifier. As an example, the nonterminal for the function *Make-A-list* above will be given the following production:

$$
\begin{array}{l}
\mathit{Make\text{-}A\text{-}list} \rightarrow \mathit{atomS} \mid (n1 \,.\, \mathit{Make\text{-}A\text{-}list}) \\
n1 \rightarrow \mathit{atomS} \mid (S \,.\, D)
\end{array}
$$

where atomS denotes a static atom, S denotes any static value and D denotes any dynamic value. The meaning of this grammar is that *Make-A-list* returns a list (terminated by a static atom) of elements of the form $n1$, which is either a static atom or a pair of a static and a dynamic value. The elements can actually never be atomic, but restrictions in the form of the productions make it necessary to include atomS in the production.

In section 7.3 the result from the binding time analysis is used to define a *program division* [Jones 88], a set of functions that will project the values of partially static variables into their static and dynamic components or reconstruct the total values from the components. The functions in the program division will be symbolically composed with the functions in the program to obtain versions with strong separation of binding times.

An example of both binding time analysis and transformation of a simple function is then presented followed by a conclusion.

7.1 Binding time descriptions

The set of values a variable can obtain during normal evaluation can be described by the regular tree-grammar:

$$Value \rightarrow atom \mid (Value\ .\ Value)$$

where atom denotes the set of all atomic values, including nil. The values are thus S-expressions as known from LISP. The idea behind the binding time analysis is to make variants of the above grammar where various parts are annotated with the binding time of the values produced. We will do this by introducing a symbol D, denoting completely dynamic values, $atomS$ denoting static atoms and S denoting all completely static values. Note that S can be defined as:

$$S \rightarrow atomS \mid (S\ .\ S)$$

We will, however, use S as a terminal symbol. The symbol D can not be defined by a production in a similar way, as the vertical bar will only be used to denote a choice (between atomic or structured values) that is decidable at static time. A small example of a binding time grammar was shown in the introduction. We will restrict right hand sides of the productions in the grammar to be either D, or of the form

$$atomS \mid (N_1\ .\ N_2)$$

where the N_i's are either S, D or non-terminals. The productions will denote sets of trees (using D and $atomS$ as terminals), each describing a binding time pattern. We define a partial order ($\subseteq_t$) of the trees by their binding time: more static parts give a greater value. The partial order is the reflexive and transitive closure of:

$$D \subseteq_t \text{ any tree}$$
$$(T_1\ .\ T_2) \subseteq_t (T_3\ .\ T_4) \text{ if } T_1 \subseteq_t T_3 \text{ and } T_2 \subseteq_t T_4$$

This ordering is opposite of the ordering used in chapter 6, where a dynamic value was the top element. We use this ordering, as it will be more natural when comparing the divisions generated from the grammar. For this reason we will order the productions by the upwards closures (rather than the downwards closures) of the sets of trees they produce:

$$P_1 \subseteq_p P_2 \text{ if } \forall T \in P_1 : \exists T_2 \in P_2,\ T_2 \subseteq_t T$$

which is the Egli-Milner ordering on the upwards closures of the sets. Note that, while this defines a partial order on upwards closed sets, it only defines a preorder on the productions, as two different productions can produce the same set of trees.

The plan is to have a non-terminal in the grammar for each variable and function in the program being analyzed, but due to the restrictions on the forms of productions, we will, as we shall see below, need extra non-terminals in addition to these. In fact, we will start by producing a grammar with fewer restrictions, using a fixed finite set of non-terminals determined by the program being analyzed. This grammar is then transformed to the form described above. This transformation may introduce new non-terminals, but still only a finite number.

The syntax of right-hand sides of the less restricted productions is:

rhs = *'D'* | *'atomS'* *'|'* *'('* *non-terminals* *'.'* *non-terminals* *')'*

non-terminals = *'D'* | *non-terminal-list*

non-terminal-list = *non-terminal* | *non-terminal* *'|'* *non-terminal-list*

The difference from the above form is that lists of alternative non-terminals are allowed inside the pairs. This is to make some operations during the analysis easier (mainly computation of the greatest lower bound of two right hand sides). In chapter 6 we had only single non-terminals inside pairs, but allowed several alternative pairs. The new form is used for two reasons: it is easier to transform to the restricted form, and the domain is smaller, making fixed-point iteration faster.

We again (pre-) order the productions by the upwards closures of the sets of trees they produce, using from now on $\subseteq$ to mean $\subseteq_p$. We want to assume two right-hand sides are ordered unless we can prove otherwise. Thus we give rules for disproving ordering:

$x \neq D \Rightarrow \neg(x \subseteq D)$ for any right-hand side x
$\neg(atomS \mid (A \,.\, B) \subseteq atomS \mid (C \,.\, E))$ if
$(A \neq D \wedge C = D \vee \exists$ non-terminal Ai in $A : \forall$ non-terminals Cj in $C : \neg(Ai \subseteq Cj))$
$\vee$
$(B \neq D \wedge E = D \vee \exists$ non-terminal Bi in $B : \forall$ non-terminals Ej in $E : \neg(Bi \subseteq Ej))$

We use $\cong$ for equivalence of non-terminals and right-hand sides: $A \cong B \Leftrightarrow A \subseteq B \wedge B \subseteq A$.

It is easy to verify that these rules correspond to the described ordering. The rules will be used to recognize fixed points during binding time analysis. It is possible to define a greatest-lower-bound ($\cap$) of two right-hand sides, giving a representative for the equivalence class of right-hand sides that describes the union of the (upwards closures of the) sets of trees produced by the two right-hand sides:

$$D \cap x = D \text{ for any right-hand side } x$$
$$atomS \mid (A \,.\, B) \cap atomS \mid (C \,.\, E) = atomS \mid (F \,.\, G)$$
where
$F =$ (if $A = D$ or $C = D$ then D else $A \mid C$)
$G =$ (if $B = D$ or $E = D$ then D else $B \mid E$)

It is assumed that operations like $A \mid C$ remove duplicated non-terminals from the concatenation of A and C.

As mentioned above, we will use variable and function names as non-terminals. As there are no local scopes for non-terminals, variables should have unique names. Otherwise they could share non-terminals, which could result in a less precise binding time classification.

It is however not enough to use variable and function names for non-terminals. The reason is that the right-hand sides must have (lists of) non-terminals inside their pairs, so it is necessary to ensure that whenever a pair is constructed there must be non-terminals to describe the arguments. This is done here by annotating the arguments of a pairing operator with non-terminals and using these to describe the binding time of their value. In the chapter 6 we solved the problem by restricting the forms of the argument expressions to the pairing operator. By using annotations we avoid the need for transforming the program in an non-invertible way.

The fixed-point iteration starts by assigning all non-terminals the completely static top value / right-hand side: $atomS \mid (S \,.\, S)$, except the non-terminals corresponding to the parameters of the goal-function which are given values describing their binding times. Then it iterates an abstract evaluation, using greatest lower bound to update non-terminals until a fixed-point is reached.

7.2 Binding time analysis

We will now describe binding time analysis of first order recursion equations using these grammars. We will use a variant of the one presented in section 3.1. This differs from the language in chapter 6 mainly by not having a *let*-expression. There are no fundamental reasons for excluding the *let*-expression, it is only to keep the following transformation algorithms relatively short.

For compactness of notation we will assume that the grammar is globally accessible using $n \rightarrow rhs$. We will also imperatively update the grammar during analysis, essentially doing a collecting interpretation.

Given recursion equations of the form:

$$\begin{aligned}&(f_1\ (X_1 1, \ldots, X_{1n_1}) = exp_1)\\ &\vdots\\ &(f_m\ (X_m 1, \ldots, X_{mn_m}) = exp_m)\end{aligned}$$

binding time analysis proceeds by repeatedly reevaluating all expressions exp_i, using the abstract evaluation E[exp_i] given below until the grammar stabilizes. It will do so, as all changes are monotonic and there are only finitely many non-terminals. The method used is essentially the same as in chapter 6.

In the rules for *CONS* and *CALL* the parameters are evaluated and, before the result is passed on, the corresponding non-terminals are updated by taking the greatest lower bound of the old and the new value.

$$E[X_{ij}] = rhs \quad \text{where } X_{ij} \rightarrow rhs$$

$$E[(QUOTE\ c)] = atomS \mid (S\ .\ S)$$

$$\begin{aligned}E[(CAR\ e1)] = {} & D \text{ if } E[e1] = D,\\ & \cap Ai,\ Ai \text{ in } A \text{ if } E[e1] = atomS \mid (A\ .\ B)\end{aligned}$$

$$\begin{aligned}E[(CDR\ e1)] = {} & D \text{ if } E[e1] = D,\\ & \cap Bi,\ Bi \text{ in } B \text{ if } E[e1] = atomS \mid (A\ .\ B)\end{aligned}$$

$$\begin{aligned}E[(ATOM\ e1)] = {} & D \text{ if } E[e1] = D,\\ & atomS \mid (S\ .\ S) \text{ otherwise}\end{aligned}$$

$$\begin{aligned}E[(EQUAL\ e1\ e2)] = {} & atomS \mid (S\ .\ S) \text{ if } E[e1] = E[e2] = atomS \mid (S\ .\ S)\\ & D \text{ otherwise}\end{aligned}$$

$$\begin{aligned}E[(CONS\ (n1 : e1)\ (n2 : e2)) = {} & atomS \mid (n1'\ .\ n2') \text{ where}\\ & n1' = D \text{ if } n1 \rightarrow D,\\ & \quad S \text{ if } n1 \rightarrow atomS \mid (S\ .\ S)\\ & \quad n1 \text{ otherwise}\\ & n2' = D \text{ if } n2 \rightarrow D,\\ & \quad S \text{ if } n2 \rightarrow atomS \mid (S\ .\ S)\\ & \quad n2 \text{ otherwise}\end{aligned}$$

$$E[(IF\ e1\ e2\ e3)] = D \text{ if } E[e1] = D$$
$$E[e2] \cap E[e3] \text{ otherwise}$$

$$E[(CALL\ f\ e1 \ldots en)] = rhs \text{ where } f \rightarrow rhs$$

7.2.1 Transformation of grammar to restricted form

The transformation to the restricted form is straightforward: a list of non-terminals in a pair is replaced by a new non-terminal whose right-hand side is the greatest lower bound of the right-hand sides of the non-terminals in the list. This new right-hand side may again contain lists of non-terminals in its pair, so the process is repeated. If a record is kept of which new non-terminals correspond to which sets of old non-terminals and re-using these, the process will terminate, as there is a finite number of such sets.

We will, during the transformation described below, sometimes have to test if a non-terminal is less than another by the ordering described above. Rather than generate the sets of trees and compare these, we will compare the definitions instead. Due to the recursion in the definitions, this may require a proof by induction. This can be done by this simple algorithm:

$$less\text{-}than(N1, N2) = less1(N1, N2, \emptyset)$$

$less1(N1, N2, ordered) =$
  if $N1 = N2$ or $N1 = D$ or $N2 = S$ or $(N1, N2) \in ordered$ then $true$
  else if $N2 = D$ then $false$
  else $less1(A1, A2, (N1, N2) \cup ordered)$ and $less1(B1, B2, (N1, N2) \cup ordered)$
    where $N1 \rightarrow atomS \mid (A1\ .\ B1)$, $N2 \rightarrow atomS \mid (A2\ .\ B2)$

The variable *ordered* contains pairs that by the induction hypothesis are assumed to be ordered.

7.3 Program divisions

The point of bringing the result in this form is that we are now able to define a program division [Jones 88]. A *program division* assigns to each variable and function a *division triple* $(\mathcal{S}, \mathcal{D}, \mathcal{P})$. $\mathcal{S}$ is a static projection function, $\mathcal{S}\ v$ discards the parts of v that are dynamic according to the division, leaving only the static part. $\mathcal{D}\ v$ similarly discards the static parts of v, leaving the dynamic part. $\mathcal{P}$ is a pairing function: if v_s and v_d are the static and dynamic part of v, $\mathcal{P}\ v_s\ v_d$ will be v. In other words: $\mathcal{P}\ (\mathcal{S}\ v)\ (\mathcal{D}\ v) = v$ for any division $(\mathcal{S}, \mathcal{D}, \mathcal{P})$ and any value v.

A program division is called *congruent* if the static part (as described by $\mathcal{S}$ in the division triple) of any function result is dependent only on the static parts of the parameters. The binding time analysis ensures that the program division constructed from the grammar as described below is congruent.

The projection functions follow the principle of [Launchbury 88], a projection replaces parts of a structure by $\perp$. In this context $\perp$ means roughly "don't know, don't need, don't care", and could be represented by any value. We will define a partial order of values with $\perp$'s as (the reflexive and transitive closure of):

$$\perp \prec \text{ any value}$$
$$(a\ .\ b) \prec (c\ .\ d) \text{ if } a \prec c \text{ and } b \prec d$$

so $a \prec b$ means a is less static than b.

We will construct the division triples from the binding time grammar by using the right hand side of the production that corresponds to a given variable or function. We will construct the division so it is congruent, and such that if we for non-terminals $X1$ and $X2$ have $X1 \subseteq X2$, then their respective static projections $\mathcal{S}_1$ and $\mathcal{S}_2$ will have the property: $\forall v : \mathcal{S}_1 v \prec \mathcal{S}_2 v$.

An additional requirement that is needed for the transformation presented later is that given $X1 \subseteq X2$, their divisions $(\mathcal{S}_1, \mathcal{D}_1, \mathcal{P}_1)$ and $(\mathcal{S}_2, \mathcal{D}_2, \mathcal{P}_2)$ must have the property that:

$$\mathcal{S}_1\ (\mathcal{P}_2\ A\ B)\ =\ \mathcal{S}_1\ A \text{ for any values } A \text{ and } B$$

which is possible, as $\mathcal{S}_1$ discards a larger part of its argument than $\mathcal{S}_2$ does. For the binding time values S and D the divisions are fairly simple: $S \sim (\lambda x.x, \lambda x.\bot, \lambda xs.\lambda xd.xs), D \sim (\lambda x.\bot, \lambda x.x, \lambda xs.\lambda xd.xd)$, using $n \sim (\mathcal{S}, \mathcal{D}, \mathcal{P})$ to mean that $(\mathcal{S}, \mathcal{D}, \mathcal{P})$ is the division triple corresponding n, where n is a non-terminal or a right hand side. We will from now on use non-terminals and their corresponding right hand sides completely interchangeably.

For more complex binding times: $A \rightarrow atom S \mid (B\ .\ C)$ the divisions are:

$$\begin{aligned}
&(\mathcal{S}_a, \mathcal{D}_a, \mathcal{P}_a)\ = (\lambda x\ .\ atom(x) \Rightarrow x \mid\mid (\mathcal{S}_b\ car(x))\ ::\ (\mathcal{S}_c\ cdr(x))\ ,\\
&\lambda x\ .\ atom(x) \Rightarrow \bot \mid\mid (\mathcal{D}_b\ car(x))\ ::\ (\mathcal{D}_c\ cdr(x))\ ,\\
&\lambda xs\ .\ \lambda xd\ .\ atom(xs) \Rightarrow xs \mid\mid (\mathcal{P}_b\ car(xs)\ car(xd))\ ::\ (\mathcal{P}_c\ cdr(xs)\ cdr(xd))\ )
\end{aligned}$$

where :: is an infix *cons*, and $a \Rightarrow b \mid\mid c$ is a conditional meaning "if a then b else c". The divisions of the non-terminals inside the pairs are called with the corresponding parts of the value as arguments. Now the requirement of having only single non-terminals inside pairs becomes clear: otherwise we could not make so simple projection functions.

[Launchbury 88] used least-upper-bound in the ordering of values with $\bot$ as a universal pairing function, but here we use explicit pairing functions for each division in order to achieve better reduction.

7.4 Transformation

The separation transformation of the program consists of constructing two new function definitions to replace each function definition in the program: given the definition of a function f:

$$f(x_1, \ldots, x_n)\ =\ exp$$

and (congruent) division triples for f and each x_i:

$$(\mathcal{S}_f, \mathcal{D}_f, \mathcal{P}_f) \text{ and } (\mathcal{S}_{x_i}, \mathcal{D}_{x_i}, \mathcal{P}_{x_i})$$

construct definitions:

$$\begin{aligned}
&f_s(x_1^s, \ldots, x_n^s)\ =\ exp_s\\
&\qquad \text{and}\\
&f_d(x_1^s, \ldots, x_n^s, x_1^d, \ldots, x_n^d)\ =\ exp_d\\
&\qquad \text{such that}\\
&\mathcal{S}_f\ Apply(f, v_1, \ldots, v_n)\ =\ Apply(f_s, \mathcal{S}_{x_1}\ v_1, \ldots, \mathcal{S}_{x_n}\ v_n)\\
&\qquad \text{and}\\
&\mathcal{D}_f\ Apply(f, v_1, \ldots, v_n)\ =\ Apply(f_d, \mathcal{S}_{x_1}\ v_1, \ldots, \mathcal{S}_{x_n}\ v_n, \mathcal{D}_{x_1}\ v_1, \ldots, \mathcal{D}_{x_n}\ v_n)
\end{aligned}$$

where $Apply(f, v_1, \ldots, v_n)$ is the result of applying the function defined by f to the arguments $v_1, \ldots, v_n$. The relation ensures that we can safely use f_s to evaluate the static part of f's value, and f_d to evaluate the dynamic part. The equalities are dependent on termination: f_s may well terminate even if f does not. Another formulation of the relation is:

$$\begin{aligned} f_s \circ \mathcal{S}_{x_1 \ldots x_n} &= \mathcal{S}_f \circ f \\ f_d \circ [\mathcal{S}_{x_1 \ldots x_n}, \mathcal{D}_{x_1 \ldots x_n}] &= \mathcal{D}_f \circ f \end{aligned}$$

using [,] to mean pairing of functions in FP style. $\mathcal{S}_{x_1 \ldots x_n}$ is a projection function for the tuple $x_1 \ldots x_n$ of arguments to f. The notation is slightly imprecise as the $\mathcal{S}$'s and $\mathcal{D}$'s are functions in the mathematical sense whereas f is an identifier representing some program text.

As f_s is completely static, it can be completely eliminated at compile time (*e.g.*, by partial evaluation), and similarly partial evaluation of f_d results in complete elimination of the parameters x_i^s. That a f_d is possible to construct is clear, as f_d is given all information about the parameters. It can be defined as:

$$f_d = \mathcal{D}_f \circ f \circ \mathcal{P}_{x_1 \ldots x_n}$$

f_s is possible because we required the program division to be congruent: the static part of f's result is only dependent on the static part of f's arguments. Since $\mathcal{S}_f \circ f$ is only dependent on the static part of the input, we have $\mathcal{S}_f \circ f = \mathcal{S}_f \circ f \circ \mathcal{S}_{x_1 \ldots x_n}$ and thus:

$$f_s = \mathcal{S}_f \circ f$$

Even though the grammar only provides division triples for the expressions that are annotated with non-terminals, a division triple for any expression can be constructed from productions found by a variant of the abstract evaluation above. No fixed point iteration is needed as the grammar already is a fixed point. $BT[e]\gamma$ is used in the transformation below to indicate the binding time of the expression e with respect to the grammar γ.

7.4.1 Transformation algorithms

We will now present algorithms that will construct e_s and e_d from the expression e and division triples for the result and variables of e, essentially by symbolically composing the projection/pairing functions with the original definition as described above. The division triples of the variables are given by supplying the complete grammar, and the division triple for the result is given by a non-terminal. The transformations try to reduce the expressions as much as they can, by removing sub-expressions that won't be needed to produce the result. To help with this a new non-terminal symbol "?" is introduced meaning "not needed". The way it is used is explained in the notes below. $S[e]\partial\gamma$ returns an e_s defining f_s given the expression e defining f, a non-terminal ∂ and a grammar γ, and $D[e]\partial\gamma$ returns an e_d defining f_d. The non-terminal ∂ represents the division of the expression result, and is initially set to the division for the function result: $\partial \sim (\mathcal{S}_f, \mathcal{D}_f, \mathcal{P}_f)$. Basically $S[e]\partial\gamma$ composes the $\mathcal{S}$ function in γ with the expression e. The grammar γ is used when converting the non-terminals to divisions. The definition of $S[]$ is followed by a list of comments that clarify various points in the definition. The numbers at the left of the lines refer to the comments.

$S[e]\partial\gamma$ =

if $\partial = D$ or $\partial =?$ then $(QUOTE\ \bot)$ /* constant expression returning $\bot$ */
else case e of

2	x_i	: if $\partial \propto x_i$ then x_i^s
3,4		else $(CALL\ S\ x_i^s)$ where $\partial \sim (S, \mathcal{D}, \mathcal{P})$
	$(QUOTE\ c)$	: $(QUOTE\ S\ c)$ where $\partial \sim (S, \mathcal{D}, \mathcal{P})$
1	$(CAR\ e')$	: $(CAR\ S[e']\partial'\gamma')$
2,5		where $\gamma' = \gamma + \partial' \rightarrow atomS \mid (\partial\ .\ ?)$
1	$(CDR\ e')$	: $(CDR\ S[e']\partial'\gamma')$
2,5		where $\gamma' = \gamma + \partial' - atomS \mid (?\ .\ \partial)$
1	$(ATOM\ e')$	: $(ATOM\ S[e']\partial'\gamma')$
2,5		where $\gamma' = \gamma + \partial' \rightarrow atomS \mid (?\ .\ ?)$
6	$(EQUAL\ e'\ e'')$	: $(EQUAL\ S[e']S\gamma\ S[e'']S\gamma)$
1	$(CONS\ e'\ e'')$	: $(CONS\ S[e']\partial'\gamma\ S[e'']\partial''\gamma)$
		where $\partial \rightarrow atomS \mid (\partial'\ .\ \partial'')$
1	$(IF\ e'\ e''\ e''')$	: $(IF\ S[e']\partial'\gamma'\ S[e'']\partial\gamma\ S[e''']\partial\gamma)$
2,5		where $\gamma' = \gamma + \partial' - atomS \mid (?\ .\ ?)$
7,2	$(CALL\ f\ e_1 \ldots e_n)$	: if $\partial \propto f$ then
		$(CALL\ f_s\ S[e_1]x_1\gamma \ldots S[e_n]x_n\gamma)$
		else
3,4		$(CALL\ S(CALL\ f_s\ S[e_1]x_1\gamma \ldots S[e_n]x_n\gamma))$
		where x_i are the formal parameters of f
		and $\partial \sim (S, \mathcal{D}, \mathcal{P})$

There are several things to note about this definition:

1 The nonterminal ∂ (representing a division triple) is used as a context [Hughes 88]: it describes how much is needed. Thus we can evaluate the argument of $ATOM$ in a context where we only need to know the head-normal form. Similarly contexts are moved inwards through CAR, CDR, $CONS$ and IF.

2 We use the ? non-terminal to mark not needed values. A D value will not be needed either, but there is a difference: a D value must evaluate to $\bot$ to ensure correctness, whereas a ? value could be anything, as it will not show in the final result. We use a relation $\propto$ extending $\cong$: $A \propto B$ if $A =?$ or $A \cong B$ or $A1 \propto B1$ and $A2 \propto B2$ where $A \rightarrow atomS \mid (A1\ .\ A2), B \rightarrow atomS \mid (B1\ .\ B2)$. $\propto$ is a asymmetric relation, ?'s can only occur on the left of $\propto$. The rules for variables and function calls can use this relation to avoid adding unnecessary projection functions. The division triple for ? is: $? \sim (\lambda x.\bot, \lambda x.\bot, \lambda xs.\lambda xd.\bot)$.

3 When a variable x_i has a division triple different from that which correspond to ∂, we should actually in e_s use $(CALL\ S\ (CALL\ \mathcal{P}_{x_i}\ x_i^s\ x_i^d))$, but since $\partial \sqsubseteq x_i$ by the congruence of the program division, we can safely replace it by $(CALL\ S\ x_i^s)$, using the property about congruent division triples. A similar case holds for function calls.

4 It is sometimes necessary to let some projection functions explicitly remain in the transformed expressions. The definitions of these functions must be added to the functions in the program unless they are simple enough to be expanded in place.

5 In the rules for *CAR*, *CDR*, *ATOM* and *IF*, new non-terminals ∂' are added to the grammar. The new non-terminals must be distinct from any previous.

6 Since completely dynamic cases are caught by the initial test, we can assume when transforming *EQUAL* that the result of equal is static, which is only the case if both parameters are completely static. Thus the context S is used in both branches.

7 Often some of the parameters of a function definition will not be used in the transformed version, it is, *e.g.*, always the case if a parameter is completely dynamic. Unused parameters can be removed from the parameter list in both the functions definition and in all calls to it.

The definition of $D[e]\partial\gamma$ is more complex: though it might seem tempting to discard static sub-expressions "as they are found in e_s", this is not always correct since they may very well be needed to find the dynamic part of the result. It is however possible to eliminate the complete expression if it is in a context where it will definitely not be needed to find the dynamic part of the value. We thus still use ∂ as a context, but a smaller context now means that more data is needed rather than less. We must also take care that the division triple we use as the context of an expression is in fact smaller than the division triple found for that expression in the binding time analysis.

Here it is sometimes necessary to let some compositions of projection and pairing functions explicitly remain in the transformed expressions. Due to the simple structure of the projection functions it is straightforward to make an efficient symbolic composition of the pairs $(\mathcal{D} \circ \mathcal{P}')$. The definitions of these new functions must be added to the functions in the program unless they are simple enough to be expanded in place. Often the combined function uses only one of its arguments, in which case the other argument can be removed.

$D[e]\partial\gamma$ =

if $\partial = S$ or $\partial =?$ then $(QUOTE\ \bot)$

else case e of

x_i : if $\partial \propto x_i$ then x_i^d
else $(CALL\ (\mathcal{D} \circ \mathcal{P}_{x_i})\ x_i^s\ x_i^d)$
where $\partial \sim (\mathcal{S}, \mathcal{D}, \mathcal{P})$
and $x_i \sim (\mathcal{S}_{x_i}, \mathcal{D}_{x_i}, \mathcal{P}_{x_i})$

$(QUOTE\ c)$: $(QUOTE\ \mathcal{D}\ c)$ where $\partial \sim (\mathcal{S}, \mathcal{D}, \mathcal{P})$

$(CAR\ e')$: $(CAR\ D[e']\partial'\gamma')$
where $\partial' = D, \gamma' = \gamma$ if $\partial = D$,
$\gamma' = \gamma + \partial' \rightarrow atomS \mid (\partial\ .\ ?)$ otherwise

$(CDR\ e')$: $(CDR\ D[e']\partial'\gamma')$
where $\partial' = D, \gamma' = \gamma$ if $\partial = D$,
$\gamma' = \gamma + \partial' \rightarrow atomS \mid (?\ .\ \partial)$ otherwise

$(ATOM\ e')$: $(ATOM\ D[e']D\gamma)$ if $BT[e']\gamma = D$,
$(ATOM\ S[e']\partial'\gamma')$ otherwise
where $\gamma' = \gamma + \partial' - atomS \mid (?\ .\ ?)$

$(EQUAL\ e'\ e'')$: $(EQUAL\ D[e']D\gamma\ D[e'']D\gamma)$

$(CONS\ e'\ e'')$: $(CONS\ D[e']\partial'\gamma\ D[e'']\partial''\gamma)$
where $(\partial', \partial'') =$
if $\partial = D$ then (D, D)
else (A, B) where $\partial \rightarrow atomS \mid (A\ .\ B)$

$(IF\ e'\ e''\ e''')$: $(IF\ D[e']D\gamma\ D[e'']\partial\gamma\ D[e''']\partial\gamma)$ if $BT[e']\gamma = D$
$(IF\ S[e']\partial'\gamma'\ D[e'']\partial\gamma\ D[e''']\partial\gamma)$ otherwise
where $\gamma' = \gamma + \partial' - atomS \mid (?\ .\ ?)$

$(CALL\ f\ e_1 \ldots e_n)$: if $\partial \propto f$ then
$(CALL\ f_d\ S[e_1]x_1\gamma \ldots S[e_n]x_n\gamma\ D[e_1]x_1\gamma \ldots D[e_n]x_n\gamma)$
else
$(CALL\ (\mathcal{D} \circ \mathcal{P}_f)\ (CALL\ f_s\ S[e_1]x_1\gamma \ldots S[e_n]x_n\gamma)$
$(CALL\ f_d\ S[e_1]x_1\gamma \ldots S[e_n]x_n\gamma\ D[e_1]x_1\gamma \ldots D[e_n]x_n\gamma))$
where x_i are the formal parameters of f
and $\partial \sim (\mathcal{S}, \mathcal{D}, \mathcal{P}), f \sim (\mathcal{S}_f, \mathcal{D}_f, \mathcal{P}_f)$

Using $S[]$ and $D[]$ to transform functions, we obtain a new version of the program, where binding times are separated. This is an example of a *staging transformation* [Jorring *et al.* 86]. If the parameters and result of the goal function of a program are completely static or dynamic, then the new program will have the same input/output function as the original. If not, then the projection functions for the argument and the pairing function for the result must be used when calling the goal function from outside.

7.5 Example

A detailed example of binding time analysis and transformation of a single function is given below. The function, which is the same as used in the introduction, constructs a list of pairs from a list of names and a list of values:

```
(Make-A-list (names values) =
 (IF (ATOM names) (QUOTE nil)
  (CONS (CONS (CAR names) (CAR values))
        (CALL Make-A-list (CDR names) (CDR values))
)))
```

For binding time analysis we annotate the parameters of *CONS*-expressions with non-terminals:

```
(Make-A-list (names values) =
 (IF (ATOM names) (QUOTE nil)
  (CONS n1 : (CONS n2 : (CAR names) n3 : (CAR values))
           Make-A-list : (CALL Make-A-list (CDR names) (CDR values))
)))
```

Note that we can use *Make-A-list* as one of the added non-terminals to the inner *CONS*. As we assume the list of names is static and the list of values is dynamic, we start with the following grammar:

$$
\begin{aligned}
&names \rightarrow atomS \mid (S\ .\ S)\\
&values \rightarrow D\\
&Make\text{-}A\text{-}list \rightarrow atomS \mid (S\ .\ S)\\
&n1 \rightarrow atomS \mid (S\ .\ S)\\
&n2 \rightarrow atomS \mid (S\ .\ S)\\
&n3 \rightarrow atomS \mid (S\ .\ S)
\end{aligned}
$$

After one iteration we get:

$$
\begin{aligned}
&names \rightarrow atomS \mid (S\ .\ S)\\
&values \rightarrow D\\
&Make\text{-}A\text{-}list \rightarrow atomS \mid (S|n1\ .\ S)\\
&n1 \rightarrow atomS \mid (S\ .\ D)\\
&n2 \rightarrow atomS \mid (S\ .\ S)\\
&n3 \rightarrow D
\end{aligned}
$$

Note that the completely static and dynamic values of $n2$ and $n3$ are propagated into the right-hand side of $n1$. The next iteration yields:

$$
\begin{aligned}
&names \rightarrow atomS \mid (S\ .\ S)\\
&values \rightarrow D\\
&Make\text{-}A\text{-}list \rightarrow atomS \mid (S|n1\ .\ S|Make\text{-}A\text{-}list)\\
&n1 \rightarrow atomS \mid (S\ .\ D)\\
&n2 \rightarrow atomS \mid (S\ .\ S)\\
&n3 \rightarrow D
\end{aligned}
$$

which is a fixed point. In this case transformation to restricted form is easy as we can use the fact $S \cap X = X$ to get:

$$
\begin{aligned}
&names \rightarrow atomS \mid (S\ .\ S)\\
&values \rightarrow D\\
&Make\text{-}A\text{-}list \rightarrow atomS \mid (n1\ .\ Make\text{-}A\text{-}list)\\
&n1 \rightarrow atomS \mid (S\ .\ D)\\
&n2 \rightarrow atomS \mid (S\ .\ S)\\
&n3 \rightarrow D
\end{aligned}
$$

Using $S[exp]Make\text{-}A\text{-}list\ \gamma$, where γ is the grammar shown above we get the static function:

$$
\begin{aligned}
&(Make\text{-}A\text{-}list_s\ (names^s\ values^s)\ =\\
&\ (IF\ (ATOM\ names^s)\ (QUOTE\ nil)\\
&\quad (CONS\ (CONS\ (CAR\ names^s)\ (QUOTE\ \bot))\\
&\qquad\qquad (CALL\ Make\text{-}A\text{-}list_s\ (CDR names^s)\ (QUOTE\ \bot))\\
&)))
\end{aligned}
$$

We can remove the parameter $values^s$, as it is never used. The dynamic part of the function is obtained by $D[exp]Make\text{-}A\text{-}list\ \gamma$:

$$
\begin{aligned}
&(Make\text{-}A\text{-}list_d\ (names^s\ values^s\ names_d\ values^d)\ =\\
&\ (IF\ (ATOM\ names^s)\ (QUOTE\ \bot)\\
&\quad (CONS\ (CONS\ (QUOTE\ \bot)\ (CAR\ values^d))\\
&\qquad\qquad (CALL\ Make\text{-}A\text{-}list_d\ (CDR\ names^s)\ (QUOTE\ \bot)\ (QUOTE\ \bot)\ (CDR\ values^d))\\
&)))
\end{aligned}
$$

The parameters $values^s$ and $names^d$ can be removed. The reduced versions of both functions are:

$$
\begin{aligned}
&(Make\text{-}A\text{-}list_s\ (names^s)\ =\\
&\ (IF\ (ATOM\ names^s)\ (QUOTE\ nil)\\
&\quad (CONS\ (CONS\ (CAR\ names^s)\ (QUOTE\ \bot))\\
&\qquad\qquad (CALL\ Make\text{-}A\text{-}list_s\ (CDR\ names^s))\\
&)))
\end{aligned}
$$

$$
\begin{aligned}
&(Make\text{-}A\text{-}list_d\ (names^s\ values^d)\ =\\
&\ (IF\ (ATOM\ names^s)\ (QUOTE\ \bot)\\
&\quad (CONS\ (CONS\ (QUOTE\ \bot\ )(CAR\ values^d))\\
&\qquad\qquad (CALL\ Make\text{-}A\text{-}list_d\ (CDR\ names^s)\ (CDR\ values^d))\\
&)))
\end{aligned}
$$

In this example it was not necessary to use $\mathcal{S}$ or $(\mathcal{D} \circ \mathcal{P})$ projections on variables or function calls in the transformations. This is the typical case, but such explicit references can occur *e.g.* in places where a partially static value is used in a context where a completely dynamic value is expected.

7.6 Conclusion

We have presented a method for separating binding times in a program. The method uses two phases: first a binding time analysis is used to obtain a program division, then this is used to transform the program.

When used in a partial evaluation system this transformation is done prior to function specialization and can thus be done at compiler generation time, having no adverse affects on the speed or size of the generated compilers.

Here we used a grammar based method for doing binding time analysis for an untyped language, but other ways of obtaining program divisions could be used. Launchbury's domain projections ([Launchbury 88] and chapter 8) would be suitable for typed lambda calculus.

Both the static and dynamic parts of a value can contain some unnecessary $\perp$'s. These can in some cases be removed by *variable splitting* ([Sestoft 86], [Romanenko 88], chapter 6, section 9.3). The method described in [Sestoft 86] uses user annotations to guide splitting a variable containing a list of values into separate variables for each value, in chapter 6 we used the structure of a partially static variable to define new variables for each dynamic part of this. [Romanenko 88] uses a type inference of residual programs to see where variables could be split, an idea that is further investigated in section 9.3 of this thesis. Using a combination of these ideas, the results of the binding time analysis can be used to annotate the dynamic parts of partially static variables. This information can then be used either during specialization (as in [Sestoft 86] or chapter 6) or after specialization in place of the result of the type inference in [Romanenko 88] or section 9.3.

Chapter 8

Binding Time Analysis for Polymorphically Typed Higher Order Languages

8.1 Introduction

Binding time analysis determines when the variables of a program are bound to their values. A typical distinction is *compile time* versus *run time*. When using partial evaluation for generation of compilers [Jones *et al.* 85], [Jones *et al.* 88], [Romanenko 88] explicit binding time annotations are essential, as argued in section 4 of this thesis and in [Bondorf *et al.* 88]. Nielson & Nielson argue in [Nielson 86] that binding time information is important when generating compilers from denotational semantics. Binding time analysis will be able to provide the necessary information.

Binding time analysis is essentially a dual problem to strictness analysis. Where strictness analysis finds how much of the parameters of a function is needed to produce a certain part of the result, binding time analysis finds how much of the result will be known, given which parts of the parameters are known. Experience shows that precision is more important in binding time analysis than in strictness analysis. An intuitive understanding of this can be found by considering the interdependence between functions in a program: given $f \circ g$, lets assume that a strictness analyzer fails to recognize that f is strict, meaning that g is in a lazy context. Even so, the strictness analyzer may find that g is strict, since it need not consider the case where g is not called (the code for g will not be called in that case). But if a binding time analyzer fails to recognize that the result of g is known, even if only part of the input is known, it will be assumed that f is called with an unknown argument, and will thus (very likely) have an unknown result. In this way an imprecise result will be propagated throughout the program being analyzed. Another reason for requiring better precision in binding time analysis is the purpose of the analysis: determining which computations should be done at compile time. Moving computations from compile time to run time can have disastrous effects on the efficiency of a program, whereas using lazy evaluation instead of strict evaluation has a more limited penalty.

Most previous work in binding time analysis [Jones *et al.* 85], [Jones *et al.* 88], (chapter 6), and [Romanenko 88] have used untyped first order languages. The notable exceptions are [Nielson 88] and [Schmidt 88] that uses typed lambda calculus and [Launchbury 88] that uses a

first order typed functional language. This chapter is in one way best comparable to Schmidt's and Nielson & Nielson's works as it uses a higher order language, but the method used is derived from Launchbury's use of projections (domain retracts) to describe binding times. Launchbury's paper describe the domain constructions and proves some properties about them, giving only hints as to how to actually perform the analysis. This chapter examines this problem in details, and finds that there are some non-trivial problems involved, especially in connection with recursive data types and higher order types.

The goal of this chapter is to provide an analysis with very precise results, aiming at a higher information content than the analysis of Nielson & Nielson.

Outline

In section 8.2 we present the type system and the principle of using projections for describing binding times. In section 8.3 we construct finite projection domains for each type in the system and define the greatest lower bound of projections. Section 8.4 presents the binding time analysis algorithm by constructing abstract versions of the operators in a functional language and proving these correct. Section 8.5 shows two examples of the analysis, section 8.6 discuss some implementation issues and in section 8.7 we round off with some conclusions.

8.2 Preliminaries

Since the binding time analysis is intended to be used in connection with partial evaluation of a language similar to the modern strongly typed functional languages like Lazy ML or Miranda, we will in this chapter use a simplified version of such languages, essentially typed curried combinators.

We use the Hindley/Milner type system as used in several functional languages (e.g. Lazy ML). This is a type system with sum, product, higher order types and polymorphism which we describe using an explicit fixed-point operator (μ) to construct recursive types, rather than by referencing to names in a global recursive type definition. We use this notation:

$$\tau \;=\; \mathit{void} \mid \mathit{int} \mid \ldots \mid \tau \times \tau \mid \mathit{tag}_1\,\tau_1 + \cdots + \mathit{tag}_n\,\tau_n \mid \tau \rightarrow \tau \mid \mu\,\alpha_i.\tau \mid \alpha_i$$

where α_i are type variables and *int* is the type of integers. There can be other base types. The sum is separated and the product is non-strict to allow lazy evaluation. *void* is an one-element type containing only $\perp$. Type variables bound by a μ are used for recursive types, and free type variables are used for polymorphism. The free variables are implicitly universally quantified. To restrict this to the Hindley/Milner type system (as known from *e.g.* ML), not all type formulas will be legal. Recursive types must be sum types (to lift the domain, it might be a "sum of one type"), and the injection tags used for sum types must be unique (*i.e.*, no two types can use the same tag). In addition to this we require that there must be finite polymorphism, *i.e.*, there can only be finitely many instantiations of polymorphic type variables. This is no great restriction, as extremely few programs use unbounded polymorphism. The Hindley/Milner type inferencer will only accept finitely polymorphic functions, if explicit user declarations of the types of functions are not used.

A polymorphic list can with this notation be expressed as:

$$\alpha\;\mathit{list} \;=\; \mu\,\beta : \mathit{nil}\;\mathit{void} \;+\; \mathit{cons}\;\alpha \times \beta$$

where α, being a free type variable, is used for polymorphism.

John Launchbury [Launchbury 88] used projections to describe binding times. A projection p_a is a retract, *i.e.*, a mapping from a domain A to itself such that:

$$p_a \sqsubseteq ID_A$$
$$p_a \circ p_a = p_a$$

where ID_A denotes the identity mapping on A. Projections can be used to describe information content: a projection (ID_A) that maps all elements of A to themselves, describes full information, whereas a projection ($ABSENT_A$) that maps all elements to $\perp$ will describe total lack of information. Thus it is natural to use projections to describe the amount of information available at partial evaluation time (compile time): whatever is left intact by a projection is considered static, whereas the parts mapped to $\perp$ are considered dynamic.

Binding time analysis consists of finding a projection p_b for the output of a function, given the function $f : A \rightarrow B$, and a projection p_a for its input, so that

$$p_b \circ f \circ p_a = p_b \circ f$$

This means that whatever p_b retains of the output of f is not dependent of the part of input that p_a discards. Note that we are interested in the p_b that discards as little as possible of its value, *i.e.*, the largest if we use the normal partial ordering of functions.

It might seem strange that we want to find the greatest solution, as it is normal in abstract interpretation to look for the least solution. However projections are not the same as sets of values, rather a projection describes information content. When using projections (with the normal ordering), it would seem natural in a forwards analysis to look for *the strongest post-condition* (*i.e.*, the greatest solution) and in a backwards analysis to look for the *weakest pre-condition* (*i.e.*, the least solution), as in strictness analysis using projections [Wadler *et al.* 87]. The choice between forwards and backwards analysis will depend on the problem. Strictness analysis "which parameters need to be known to find the result of the function" seems to be a kind of pre-condition, whereas the p_b above represents a post-condition for f with respect to the pre-condition represented by p_a.

Note that it will in general not be possible to find a maximal p_b. Consider the function:

$$f\ x = if\ g\ x\ then\ x\ else\ x$$

where we assume that the conditional is strict in the condition. If $g\ x$ never terminates then $f = \lambda x.\perp$, so $p_b = ID$ is safe by the equation above. If $g\ x$ sometimes terminates $p_b = p_a$ is the largest safe result. Thus, if $p_a \neq ID$ then finding the largest p_b would involve solving the halting problem! We will in our analysis assume termination of programs, so the results of the analysis will be the same for strict and lazy languages.

The result of the binding time analysis can be used by a partial evaluator, either by annotating expressions with their binding time, as in [Jones *et al.* 85] and [Jones *et al.* 88] or by transforming the program to separate the binding times, as in chapter 7.

8.3 Projections

Given a type formula for a type A. we want to construct a finite domain of representations of projections $\mathcal{P}_A$ for A. Any $\mathcal{P}_A$ should contain representations of ID_A and $ABSENT_A$, where

$ABSENT_A = \lambda x.\bot_A$. ID and $ABSENT$ will be used polymorphically to represent ID_A and $ABSENT_A$ for any A. We will assume the existence of an implicit semantic function, mapping representations to projections. We allow several representations of the same projection, but we will be able to identify these. We will in the descriptions below allow ourselves to be a bit sloppy in distinguishing projections and their representations. When the difference is important, we will state which meaning is used. We will order the representations by comparing the projections they represent.

$$\begin{aligned}\mathcal{P}_{void} &= \{ID\} = \{ABSENT\}\\ \mathcal{P}_{int} &= \{ID, ABSENT\}\\ \mathcal{P}_{A\times B} &= \{\langle f,g\rangle \mid f \in \mathcal{P}_A,\ g \in \mathcal{P}_B\}\end{aligned}$$

where $\langle f,g\rangle$ represents $\lambda(a,b).(f\ a, g\ b)$.

$$\mathcal{P}_{tag_1\ A_1 +\cdots+\ tag_n\ A_n} = \{ABSENT\} \cup \{tag'_1\ f_1 \mp \cdots \mp tag'_n\ f_n \mid f_i \in \mathcal{P}_{A_i}\}$$

Where we use $tag'_1\ f_1 \mp \cdots \mp tag'_n\ f_n$ to represent

$$\lambda x.case\ x\ of\ tag_1\ v : tag_1\ (f_1\ v); \ldots tag_n\ v : tag_n\ (f_n\ v)$$

where the *case* expression is strict in its sum-type argument (*i.e.*, $case\ \bot\ of \ldots = \bot$). Note that

$$\begin{aligned}ID_{tag_1\ A_1+\cdots+tag_n\ A_n} &= tag'_1\ ID_{A_1} \mp \cdots \mp tag'_n\ ID_{A_n}\\ ABSENT_{tag_1\ A_1+\cdots+tag_n\ A_n} &\neq tag'_1\ ABSENT_{A_1} \mp \cdots \mp tag_n\ ABSENT_{A_n}\end{aligned}$$

For recursive types we have:

$$\begin{aligned}\mathcal{P}_{\mu\ \alpha_i.A} &= \{fix\lambda f_i.p \mid p \in \mathcal{P}_A\}\\ \mathcal{P}_{\alpha_i} &= \{f_i\} \qquad \text{if } \alpha_i \text{ is bound by a } \mu\end{aligned}$$

Here we require consistency in the choice of identifiers. Free type variables have no fixed set of projections, rather they inherit the projection domain of any type that instantiates them[1]. The above definitions gives the following set of projections for the $\alpha\ list$ type:

$$\mathcal{P}_{\alpha\ list} = \{fix\lambda g.ABSENT\} \cup \{fix\lambda g.nil'\ ID \mp cons'\ \langle f,g\rangle \mid f \in \mathcal{P}_\alpha\}$$

So the projection will either return $\bot$, or map a projection onto the elements of the list.

So far we have not defined projections on functional types. A projection on a function must return a smaller function: a function that always returns a smaller value. This can be done in several ways, it e.g.,. by composing it with a projection (in either end). It is however more useful (as we will see) to think of the function as a closure, and then replace parts of the closure by $\bot$. If we have the program in named combinator form, all free variables are converted to parameters, so a closure is just a function name and a partial list of parameters. A projection ($ABSENT_{A\to B}$) can replace the complete closure by $\bot$, or another projection can, depending on which function name the closure contains, apply projections to the parameters, giving a projection of the form:

[1]This is only safe because we assume finite polymorphism. Otherwise we might generate infinitely many projections during binding time analysis.

$$
\begin{array}{l}
\lambda c.\ case\ c\ of \\
\quad [f\ v_1 \ldots v_n] : [f\ (p_1\ v_1) \ldots (p_n\ v_n)] \\
\quad \cdots \qquad\qquad : \cdots \\
\quad [g\ v_1 \ldots v_m] : [g\ (q_1\ v_1) \ldots (q_m\ v_m)]
\end{array}
$$

which can be represented by a set of abstract closures:

$$\{[f\ p_1 \ldots p_n], \ldots, [g\ q_1 \ldots q_m]\}$$

In this way the domain of projections for higher order types are not only dependent on the type, but also on the program. The set need not contain closures for all functions in the program, as it can be a default that any closure not represented will have the identity projection.

Whereas the projections for the first order values have a meaning independent of the program being analyzed, thus being extensional, the projection for functional values is strongly dependent on the actual program text, thus being very intensional. This makes the analysis less "clean", but it also greatly increases the quality of the result of the analysis. We believe that this increased precision is worth the price paid in loss of conceptual clarity.

Greatest Lower Bound

During a binding time analysis we will often want, given two or more projections, to find a projection that discards all that any of these discards and keeps all that all of these keep. This is the greatest lower bound ($\sqcap$) of the projections. The greatest lower bound of two projections p_1 and p_2 can be defined as:

$$p_1 \sqcap p_2 = \lambda x.(p_1\ x) \sqcap (p_2\ x)$$

Note that in general the greatest lower bound of two projections need not be idempotent, and thus not a projection. In the domains of representations we will, however, be able to construct a representation of a lower bound that will represent a projection, and that (with the exception of the functional types) will represent the greatest lower bound in the domains of projections as well.

We will now construct the greatest lower bound for each $\mathcal{P}_A$.

For all types and all projections p we will have:

$$
\begin{array}{c}
ID \sqcap p = p \\
ABSENT \sqcap p = ABSENT
\end{array}
$$

This completely defines $\sqcap$ for the types *void* and *int*, so we will continue with the remaining types, showing only the cases not defined by the above rules.

$\mathcal{P}_{A \times B}$:

$$\langle p_{a1}, p_{b1} \rangle \sqcap \langle p_{a2}, p_{b2} \rangle = \langle p_{a1} \sqcap p_{a2}, p_{b1} \sqcap p_{b2} \rangle$$

$\mathcal{P}_{tag_1\ A_1 + \cdots + tag_n\ A_n}$:

$$
\begin{array}{l}
tag_1\ p_{a_{11}} \mp \cdots \mp tag_n\ p_{a_{1n}} \sqcap tag_1\ p_{a_{21}} \mp \cdots \mp tag_n\ p_{a_{2n}} = \\
\qquad tag_1\ (p_{a_{11}} \sqcap p_{a_{21}}) \mp \cdots \mp tag_n\ (p_{a_{1n}} \sqcap p_{a_{2n}})
\end{array}
$$

All these are trivial. For recursive types we have:

$$\mathcal{P}_{\mu\,\alpha_i:A}\ :$$

$$fix\lambda f_i.p_{a1} \sqcap fix\lambda g_i.p_{a2} = fix\lambda h_i.(p_{a1}[f_i \backslash h_i] \sqcap p_{a2}[g_i \backslash h_i])$$

$$\mathcal{P}_{\alpha_i}\ :$$

$$h_i \sqcap h_i = h_i$$

In the rule for a bound type variable we can assume that the same identifier is used, as a preceding use of the rule for recursive types will have performed alpha-conversion to ensure this. The rule for recursive types requires that $\sqcap$ is continuous in the domains of representations of projections. It is trivially monotonous, so only the limits of chains needs to be investigated. For the finite domains this is again trivial, and for the function space projections it can be proven by induction (omitted here).

For projections on functions we have:

$$\mathcal{P}_{A\to B}\ :$$

$$p_1 \sqcap p_2 = \{cl \wedge p_2 \mid cl \in p_1\} \cup \{cl \wedge p_1 \mid cl \in p_2\}$$

otherwise

where

$$[f\ p_{11} \ldots p_{1n}] \wedge p =$$

$$[f\ (p_{11} \sqcap p_{21}) \ldots (p_{1n} \sqcap p_{2n})]$$

if there is a closure $[f\ p_{21} \ldots p_{2n}] \in p$

$$[f\ p_{11} \ldots p_{1n}]$$

otherwise

It is easy to see that the constructed projection $p_3 = p_1 \sqcap p_2$ is in $\mathcal{P}_{A\to B}$ if the arguments (p_1, p_2) are. To see that it is the greatest lower bound of these consider the result of applying p_i to a closure $[g\ v_1 \ldots v_n]$. If there is no abstract closure of the right structure in either of p_1 or p_2, there will not be in p_3 either, so all of them would map the closure to itself. If there are appropriate abstract closures $[g\ p_{11} \ldots p_{1n}]$ and $[g\ p_{21} \ldots p_{2n}]$ in p_1 and p_2 then there is an abstract closure $[g\ (p_{11} \sqcap p_{21}) \ldots (p_{1n} \sqcap p_{2n})]$ in p_3. Hence what we need to show is that

$$[g\ (p_{11}\ v_1) \ldots (p_{1n}\ v_n)] \sqcap [g\ (p_{21}\ v_1) \ldots (p_{2n}\ v_n)] = [g\ ((p_{11} \sqcap p_{21})\ v_1) \ldots ((p_{1n} \sqcap p_{2n})\ v_n)]$$

If the closure is seen as a data structure, this is certainly true. If it is seen as a function, that is not necessarily the case. If g is monotonic we have

$$[g\ (p_{11}\ v_1) \ldots (p_{1n}\ v_n)] \sqcap [g\ (p_{21}\ v_1) \ldots (p_{2n}\ v_n)] \sqsupseteq [g\ ((p_{11} \sqcap p_{21})\ v_1) \ldots ((p_{1n} \sqcap p_{2n})\ v_n)]$$

but the converse is not always true. Thus we have a lower bound, but not necessarily the greatest. It is however the greatest in the required form. Also, the functions g for which it isn't the greatest lower bound would behave similar to parallel OR, and are thus not expressible in lambda calculus. So for the language presented below, the construction will indeed yield the greatest lower bound.

8.4 Binding Time Analysis

Binding time analysis consists (as mentioned earlier) of, given a definition of a function and a projection for its input, to find a projection for its output so that

$$p_b \circ f \circ p_a \;=\; p_b \circ f$$

where p_a is the projection for the input to f and p_b is the projection for the output. As mentioned in section 2, it will not be possible to find a maximal safe p_b, so we will just try to find a safe p_b while taking care not to make it needlessly small. This will be done by constructing an abstract function $f^\#$ for each function f, such that $f^\#\ p_a \;=\; p_b$, where p_b is the greatest projection in $\mathcal{P}_B$ that has the required property. The $f^\#$ is constructed by replacing each operator *op* in f's definition by an abstract operator $op^\#$. We must for each $op^\#$ prove that it has the correct relation to *op*. We will do this by first defining the syntax, then for each operator in the expression syntax define the abstract operator, and then prove that the properties hold.

$$\begin{array}{rll} < program > ::= & f_1 \ < variable_{11} > \ldots < variable_{1n_1} > = < exp_1 >; \\ & \cdots \\ & f_m \ < variable_{m1} > \ldots < variable_{mn_m} > = < exp_m >; \end{array}$$

$$\begin{array}{rll} < exp > \ ::= & < variable > \\ \mid & < function\ name > \\ \mid & (< exp >, < exp >) \\ \mid & fst \ < exp > \\ \mid & snd \ < exp > \\ \mid & tag_i \ < exp > \\ \mid & case \ < exp > \ of & tag_1 \ < variable > : < exp >; \\ & & \cdots \\ & & tag_n \ < variable > : < exp > \\ \mid & < exp > < exp > \end{array}$$

Thus we can define $f_i^\#$ as:

$$f^\#\ p_{x_1} \cdots p_{x_{n_i}} \;=\; e_i^\#$$

where f_i is defined by

$$f_i\, x_1 \ldots x_{n_i} \;=\; e_i$$

and $e_i^\#$ is e_i where all operators have been replaced by their abstract (hashed) counterpart, all variables x_j have been replaced by projection variables p_{x_j} and all named functions f_k by a constant singleton set of abstract closures $\{[f_k]\}$.
The operators can be given polymorphic types:

$$\begin{array}{ll} (\_,\_) & : A \times B \rightarrow A \times B \\ fst & : A \times B \rightarrow A \\ snd & : A \times B \rightarrow B \\ tag_i & : A_i \rightarrow (tag_1\ A_1 + \cdots + tag_n\ A_n) \\ case & : (A_1 \rightarrow B) \times \cdots \times (A_n \rightarrow B) \rightarrow (tag_1\ A_1 + \cdots + tag_n\ A_n) \rightarrow B \\ apply & : (A \rightarrow B) \times A \rightarrow B \end{array}$$

where we look at the branches of the *case* operator as functions of the "pattern" variables. Note that the tag_i and *case* operators are not really polymorphic in the sum type arguments. This is because we required unique injection tags, so we could say we have a family of *case* functions. The branches of a *case* expression are seen as functions from the summand types to the result type. These functions are seen as a part of the *case* operator. The *apply* operator is invisible in the actual syntax, where $< exp > < exp >$ is application of a function to an argument. In addition to these, we will also have constants and operations on the base types (1,2,'a',+,-,...). We can now give types for the abstract operators:

$$
\begin{array}{ll}
(\_,\_)^{\#} & : \mathcal{P}_A \times \mathcal{P}_B \rightarrow \mathcal{P}_{A \times B} \\
fst^{\#} & : \mathcal{P}_{A \times B} \rightarrow \mathcal{P}_A \\
snd^{\#} & : \mathcal{P}_{A \times B} \rightarrow \mathcal{P}_B \\
tag_i^{\#} & : \mathcal{P}_{A_i} \rightarrow \mathcal{P}_{tag_1\ A_1 + \cdots + tag_n\ A_n} \\
case^{\#} & : (\mathcal{P}_{A_1} \rightarrow \mathcal{P}_B) \times \cdots \times (\mathcal{P}_{A_n} \rightarrow \mathcal{P}_B) \rightarrow \mathcal{P}_{tag_1\ A_1 + \cdots + tag_n\ A_n} \rightarrow \mathcal{P}_B \\
apply^{\#} & : \mathcal{P}_{A \rightarrow B} \times \mathcal{P}_A \rightarrow \mathcal{P}_B
\end{array}
$$

Note that the parameters of $case^{\#}$ corresponding to the branches are of type $\mathcal{P}_{A_i} \rightarrow \mathcal{P}_B$ rather than $\mathcal{P}_{A_i \rightarrow B}$ as one could expect. This is because the branches in $case^{\#}$ will be abstract expressions (built from abstract operators), and thus projection transformers rather than projections of value transformers. We will now give definitions of the abstract operators. The abstract version of a base type constant is the identity projection and the abstract version of a strict base type operator (like +,* etc.) returns *ABSENT* unless all parameters are *ID*, in which case it returns *ID*

$$
\begin{array}{lll}
(\_,\_)^{\#} & = \lambda(p_a, p_b).\langle p_a, p_b \rangle & \\
fst^{\#} & = \lambda p_{ab}.p_a \text{ where } p_{ab} = \langle p_a, p_b \rangle & \\
snd^{\#} & = \lambda p_{ab}.p_b \text{ where } p_{ab} = \langle p_a, p_b \rangle & \\
tag_i^{\#} & = \lambda p_a.tag_1'\ ID \mp \cdots \mp tag_i'\ p_a \mp \cdots \mp tag_n'\ ID & \\
case^{\#} & = \lambda(F^{\#}_{a_1 b}, \ldots, F^{\#}_{a_n b}).\lambda\ p_a. & \\
 & \quad case\ p_a\ of & \\
 & \qquad ABSENT & : ABSENT \\
 & \qquad tag_1'\ f_1 \mp \cdots \mp tag_n'\ f_n & : (F^{\#}_{a_1 b}\ f_1) \sqcap \cdots \sqcap (F^{\#}_{a_n b}\ f_n)
\end{array}
$$

In the abstract expressions variables (e.g. x) have been replaced by projection identifiers (e.g. p_x). Function names (e.g. f) are replaced by projection constants (e.g. {[f]}). The rules have assumed the absence of fix in the projections. When decomposing projections on recursive types we convert $\mu\,\alpha : \tau[\alpha]$ to $\tau[\mu\,\alpha : \tau[\alpha]]$ and $fix\lambda f.p[f]$ to $p[fix\lambda f.p[f]]$. As fix is used only on sum type projections, it is only $case^{\#}$ that has to worry about this. Similarly it is only $tag_i^{\#}$ that have to worry about introducing fix. We will use the fact that the tags are unique to find the type they inject into, specifically if it is a recursive type. In case it is we will extract the projection corresponding to the type variable by the method below, and create a new recursive projection by taking the greatest lower bound of this and the projection obtained by replacing the extracted parts by a projection identifier (f) using the *replace* function below, and adding a $fix\lambda f$ around the resulting projection.

$$
\begin{array}{l}
extract\ \alpha[\tau]\ p\ = \\
\quad if\ \alpha \text{ does not occur free in } \tau\ then\ ID \\
\quad else\ if\ p\ =\ ABSENT\ then\ ABSENT \\
\quad else\ case\ (\tau\ ,\ p)\ of \\
\qquad \begin{array}{ll}
(\alpha\ ,\ p) & : p \\
(\tau_1 \times \tau_2\ ,\ \langle p_1, p_2 \rangle) & : (extract\ \alpha[\tau_1]\ p_1) \sqcap (extract\ \alpha[\tau_2]\ p_2) \\
(tag_1\tau_1 + \cdots + tag_n\tau_n\ , & \\
\quad tag'_1 p_1 \mp \cdots \mp tag'_n p_n) & : (extract\ \alpha[\tau_1]\ p_1) \sqcap \cdots \sqcap (extract\ \alpha[\tau_n]\ p_n) \\
(\tau_1 \rightarrow \tau_2\ ,\ p) & : ID \\
(\mu\ \beta : \tau_1\ ,\ fix\lambda f.p_1) & : extract\ \alpha[\tau_1]\ p_1
\end{array}
\end{array}
$$

$$
\begin{array}{l}
replace\ \alpha\ f\ [\tau]\ p\ = \\
\quad if\ \alpha \text{ does not occur free in } \tau\ then\ p \\
\quad else\ if\ p\ =\ ABSENT\ then\ ABSENT \\
\quad else\ case\ (\tau\ ,\ p)\ of \\
\qquad \begin{array}{ll}
(\alpha\ ,\ p) & : f \\
(\tau_1 \times \tau_2\ ,\ \langle p_1, p_2 \rangle) & : \langle replace\ \alpha\ f\ [\tau_1]\ p_1\ ,\ replace\ \alpha\ f\ [\tau_2]\ p_2 \rangle \\
(tag_1\tau_1 + \cdots + tag_n\tau_n\ , & \\
\quad tag'_1 p_1 \mp \cdots \mp tag'_n p_n) & : tag'_1(replace\ \alpha\ f\ [\tau_1]\ p_1) \mp \cdots \\
 & \qquad \mp tag'_n(replace\ \alpha\ f\ [\tau_n]\ p_n) \\
(\tau_1 \rightarrow \tau_2\ ,\ p) & : p \\
(\mu\ \beta : \tau_1\ ,\ fix\lambda g.p_1) & : fix\lambda g.replace\ \alpha\ f\ [\tau_1]\ p_1
\end{array}
\end{array}
$$

$apply^{\#}$ must have the property:

$$apply^{\#}(p_{ab}, p_a) \circ (p_{ab}\ f) \circ p_a\ =\ apply^{\#}(p_{ab}, p_a) \circ f$$

for any function $f\ :\ A \rightarrow B$ and any projection $p_a \in \mathcal{P}_A$. If this should hold for *any imaginable* f, it is fairly easy to see that $apply^{\#}(p_{ab}, p_b)\ =\ ABSENT$ unless both p_{ab} and p_a are the identity projections (in which case it is ID). But by relaxing the condition to hold for only the functions that actually occur in a given program, we can obtain better results. For the previously mentioned projection structure we can define:

$$
\begin{array}{l}
apply^{\#}(p_{ab}, p_a)\ =\ \begin{array}{ll} ABSENT & \text{if } p_{ab}\ =\ ABSENT \\ \bigsqcap_{c \in p_{ab}}\ app_1^{\#}\ c\ p_a & otherwise \end{array} \\
\quad \text{where} \\
\qquad app_1^{\#}\ [f\ p_1 \ldots p_n]\ p_a\ =\ \begin{array}{ll} f^{\#}\ p_1 \ldots p_n\ p_a & \text{if } f \text{ has } n+1 \text{ parameters} \\ \{[f\ p_1 \ldots p_n\ p_a]\} & \text{otherwise} \end{array}
\end{array}
$$

Essentially we add parameters to all the closures involved, producing new closures. When the final element in a closure is given we use the abstract function $f^{\#}$ on the argument projections giving a result projection. Since it is possible that some of the abstract closures will be given their final argument and others not, we produce a result projection for each closure (which can be just by adding a parameter to the closure) and take the greatest lower bound of these.

This definition gives us

$$apply^{\#}(\{[f]\}, p_a)\ =\ f^{\#}\ p_a$$

for first-order functions f, as one would expect. $f^{\#}$ must be defined to handle polymorphism. This is done by letting it be polymorphic over projection domains. As mentioned above the type system ensures that there will only be finitely many instantiations of any given polymorphic type, so this will not cause non-termination.

8.4.1 Correctness

To prove the definitions of $op^{\#}$ correct, we must first prove that for an operator $op : A \to B$ that

$$p_b = op^{\#}\, p_a \;\Rightarrow\; p_b \circ op \circ p_a = p_b \circ op$$

for all p_a (the safety criterion). The abstract versions of base type constants and operators are trivially correct, so we will continue straight away with the remaining cases:

$$op = (\_,\_) : \; p_b = (\_,\_)^{\#}\,(p_{a1}, p_{a2}) = \langle p_{a1}, p_{a2}\rangle$$

$$\begin{aligned}
& p_b \circ (_,_) \circ \langle p_{a1}, p_{a2}\rangle \\
&= \langle p_{a1}, p_{a2}\rangle \circ \langle p_{a1}, p_{a2}\rangle \\
&= \langle p_{a1} \circ p_{a1}, p_{a2} \circ p_{a2}\rangle \\
&= \langle p_{a1}, p_{a2}\rangle \\
&= \langle p_{a1}, p_{a2}\rangle \circ (_,_) \\
&\quad \square
\end{aligned}$$

$$op = fst : \; p_b = fst^{\#}\, p_a = p_{a1} \text{ where } p_a = \langle p_{a1}, p_{a2}\rangle$$

$$\begin{aligned}
& p_b \circ fst \circ p_a \\
&= p_{a1} \circ fst \circ \langle p_{a1}, p_{a2}\rangle \\
&= p_{a1} \circ fst \circ (\lambda(x,y).(p_{a1}\, x, p_{a2}\, y)) \\
&= p_{a1} \circ (\lambda(x,y).p_{a1}\, x) \\
&= p_{a1} \circ p_{a1} \circ (\lambda(x,y).x) \\
&= p_{a1} \circ fst \\
&\quad \square
\end{aligned}$$

$$op = snd : \; p_b = snd^{\#}\, p_a = p_{a2} \text{ where } p_a = \langle p_{a1}, p_{a2}\rangle$$

$$\begin{aligned}
& p_b \circ snd \circ p_a \\
&= p_{a2} \circ snd \circ \langle p_{a1}, p_{a2}\rangle \\
&= p_{a2} \circ snd \circ (\lambda(x,y).(p_{a1}\, x, p_{a2}\, y)) \\
&= p_{a2} \circ (\lambda(x,y).p_{a2}\, y) \\
&= p_{a2} \circ p_{a2} \circ (\lambda(x,y).y) \\
&= p_{a2} \circ snd \\
&\quad \square
\end{aligned}$$

$$op = tag_i : p_b = tag_i^{\#}\ p_a = tag_1' ID \mp \cdots \mp tag_i' p_a \mp \cdots \mp tag_n' ID$$

$$\begin{aligned}
& p_b \circ tag_i \circ p_a \\
&= (tag_1' ID \mp \cdots \mp tag_i' p_a \mp \cdots \mp tag_n' ID) \circ tag_i \circ p_a \\
&= \left(\lambda x.case\ x\ of \left\{ \begin{array}{l} tag_1\ v : tag_1\ v; \\ \cdots \\ tag_i\ v : tag_i(p_a\ v); \\ \cdots \\ tag_n\ v : tag_n\ v \end{array} \right. \right) \circ tag_i \circ p_a \\
&= (\lambda v.tag_i\ (p_a\ v)) \circ p_a \\
&= tag_i \circ p_a \circ p_a \\
&= tag_i \circ p_a \\
&= p_b \circ tag_i \\
&\quad \square
\end{aligned}$$

For the *case* operator we will look at $case(f_1, \ldots, f_n)$ and the corresponding $case^{\#}(f_1^{\#}, \ldots, g_1^{\#})$. We will consider the functions f_i as part of the *case* operator and the abstract functions $f_i^{\#}$ as part of the $case^{\#}$ operator. We will assume by induction:

$$(f_i^{\#}\ p_a) \circ f_i \circ p_a\ =\ (f_i^{\#}\ p_a) \circ f_i$$

for all f_i and all p_a. We will make use of a lemma:

$$f \sqsubseteq g\ \Rightarrow\ f \circ g\ =\ f \quad \text{for all projections } f \text{ and } g$$

proof:

$$\begin{aligned}
& f \sqsubseteq g \\
\Rightarrow\ & f \circ f \sqsubseteq f \circ g \\
\Rightarrow\ & f \sqsubseteq f \circ g
\end{aligned}$$

$$\begin{aligned}
& g \sqsubseteq ID \\
\Rightarrow\ & f \circ g \sqsubseteq f \circ ID \\
\Rightarrow\ & f \circ g \sqsubseteq f \\
& \square
\end{aligned}$$

We will start with the case where $p_a\ =\ ABSENT$:

$$\begin{aligned}
& op\ =\ case\ (f_1, \ldots, f_n)\ : \\
& p_b\ =\ case^{\#}\ (f_1^{\#}, \ldots, f_n^{\#})\ ABSENT\ =\ ABSENT
\end{aligned}$$

$$\begin{aligned}
& p_b \circ case\ (f_1, \ldots, f_n) \circ p_a \\
&= ABSENT \circ case\ (f_1, \ldots, f_n) \circ ABSENT \\
&= ABSENT \\
&= ABSENT \circ case\ (f_1, \ldots, f_n) \\
&\quad \square
\end{aligned}$$

and then the case where $p_a\ =\ tag_1' p_{a_1} \mp \cdots \mp tag_n' p_{a_n}$:

$$
\begin{aligned}
op &= case\,(f_1,\ldots,f_n)\ : \\
p_b &= case^{\#}\,(f_1^{\#},\ldots,f_n^{\#})\,(tag_1\,p_{a_1} \mp \cdots \mp tag_n\,p_{a_n}) \\
&= (f_1^{\#}\ p_{a_1}) \sqcap \cdots \sqcap (f_n^{\#}\ p_{a_n})
\end{aligned}
$$

$$
\begin{aligned}
& p_b \circ case\,(f_1,\ldots,f_n) \circ (tag_1\,p_{a_1} \mp \cdots \mp tag_n\,p_{a_n}) \\
&= p_b \circ (\lambda x.case\ x\ of\ tag_1\ v:\ f_1\ v; \ldots tag_n\ v:\ f_n\ v) \\
&\quad \circ (\lambda x.case\ x\ of\ tag_1\ v:\ tag_1(p_{a_1}\ v); \ldots tag_n\ v:\ tag_n(p_{a_n}\ v)) \\
&= p_b \circ (\lambda x.case\ x\ of\ tag_1\ v:\ f_1(p_{a_1}\ v); \ldots tag_n\ v:\ f_n(p_{a_n}\ v)) \\
&= (\lambda x.case\ x\ of\ tag_1\ v:\ p_b \circ f_1 \circ p_{a_1}\ v; \ldots tag_n\ v:\ p_b \circ f_n \circ p_{a_n}\ v) \\
&= (\lambda x.case\ x\ of\ \ tag_1\ v:\ p_b \circ (f_1^{\#}\ p_{a_1}) \circ f_1 \circ p_{a_1}\ v; && \text{by the lemma, as } p_b \sqsubseteq (f_i^{\#}\ p_{a_i}) \\
&\qquad\qquad\quad \ldots \\
&\qquad\qquad\quad tag_n\ v:\ p_b \circ (f_n^{\#}\ p_{a_n}) \circ f_n \circ p_{a_n}\ v) \\
&= (\lambda x.case\ x\ of\ \ tag_1\ v:\ p_b \circ (f_1^{\#}\ p_{a_1}) \circ f_1\ v; && \text{by the inductive assumption} \\
&\qquad\qquad\quad \ldots \\
&\qquad\qquad\quad tag_n\ v:\ p_b \circ (f_n^{\#}\ p_{a_n}) \circ f_n\ v) \\
&= (\lambda x.case\ x\ of\ tag_1\ v:\ p_b \circ f_1\ v; \ldots tag_n\ v:\ p_b \circ f_n\ v) && \text{by the lemma again} \\
&= p_b \circ (\lambda x.case\ x\ of\ tag_1\ v:\ f_1\ v; \ldots tag_n\ v:\ f_n\ v) \\
&= p_b \circ case\,(f_1,\ldots,f_n) \\
&\quad \square
\end{aligned}
$$

Proving safety for $apply^{\#}$ is proving the property:

$$apply^{\#}(p_{ab}, p_a) \circ (p_{ab}\ f) \circ p_a\ =\ apply^{\#}(p_{ab}, p_a) \circ f$$

for any functional value $f\ :\ A \rightarrow B$ that can occur while executing the program. If the function projection p_{ab} is $ABSENT$ then $apply^{\#}(p_{ab}, p_a)$ is $ABSENT$ as well, making the property trivially true. Otherwise that f is a closure $[g\ v_1 \ldots v_n]$, and p_{ab} is a set of abstract closures. Then $p_{ab}\ f$ is $[g\ (p_1\ v_1) \ldots (p_n\ v_n]$ where $[g\ p_1 \ldots p_n]$ is the abstract closure for g in p_{ab}, or $[g\ ID \ldots ID]$ if there is no such abstract closure. $apply^{\#}(p_{ab}, p_a)$ will be less than $app_1^{\#}\ [g\ p_1 \ldots p_n]\ p_a$, so by the lemma given above, all we have to prove is that

$$
\begin{aligned}
&(app_1^{\#}\ [g\ p_1 \ldots p_n]\ p_a) \circ [g\ (p_1\ v_1) \ldots (p_n\ v_n)] \circ p_a\ = \\
&(app_1^{\#}\ [g\ p_1 \ldots p_n]\ p_a) \circ [g\ v_1 \ldots v_n]
\end{aligned}
$$

We will consider the two cases: g has $n+1$ parameters, and g has more than $n+1$ parameters. In the first case

$$app_1^{\#}\ [g\ p_1 \ldots p_n]\ p_a\ =\ g^{\#}\ p_1 \ldots p_n\ p_a$$

and

$$[g\ (p_1\ v_1) \ldots (p_n\ v_n)] \circ p_a\ =\ \lambda v_a.g\ (p_1\ v_1) \ldots (p_n\ v_n)\ (p_a\ v_a)$$

By induction we can assume that

$$(g^{\#}\ p_1 \ldots p_n\ p_a)\ (g\ (p_1\ v_1) \ldots (p_n\ v_n)\ (p_a\ v_a))\ =\ (g^{\#}\ p_1 \ldots p_n\ p_a)\ (g\ v_1 \ldots v_n\ v_a)$$

which is what we needed to prove the property. In the case of g having more than $n+1$ parameters

$$app_1^{\#}\,[g\ p_1 \ldots p_n]\ p_a \;=\; \{[g\ p_1 \ldots p_n\ p_a]\}$$

and

$$[g\ (p_1\ v_1) \ldots (p_n\ v_n)] \circ p_a \;=\; \lambda v_a.[g\ (p_1\ v_1) \ldots (p_n\ v_n)\ (p_a\ v_a)]$$

and since

$$\begin{aligned}
&\{[g\ p_1 \ldots p_n\ p_a]\}\ [g\ (p_1\ v_1) \ldots (p_n\ v_n)\ (p_a\ v_a)] \\
&= [g\ (p_1\ (p_1\ v_1)) \ldots (p_n\ (p_n\ v_n))\ (p_a\ (p_a\ v_a))] \\
&= [g\ (p_1\ v_1) \ldots (p_n\ v_n)\ (p_a\ v_a)] \\
&= \{[g\ p_1 \ldots p_n\ p_a]\}\ [g\ v_1 \ldots v_n\ v_a]
\end{aligned}$$

we have what we want.

8.4.2 Recursive function space projections

A function space projection may contain abstract closures that have parameters that again are abstract closures *etc.*, so an analysis might run infinitely by building larger and larger nested structures. To solve this we propose to approximate infinite nested structures by recursive definitions. This will be done by recognizing similarities between different levels in the closures and construct a recursive definition by identifying the levels. This might lead to a less precise (smaller) projection, but will not make a safe projection unsafe, so the correctness proofs will still be valid.

When a recursive projection is used it is unfolded one step (which will not change the value). After application the result is made recursive again if necessary.

For recursive types the type structure determines when recursion needs to be introduced in the projection. For functional types this is not so easy. However, abstract closures can only be infinitely nested if a function's arguments can contain a closure of that same function. This can to some extent be decided by the type structure, in the sense that the type of a function must allow a parameter that contain a function of the same type. There are several ways of tying the recursion. One is to make a recursive definition every time the type would make it possible, another is to do it when a closure actually occur nested inside one of the same structure. The tighter we tie the recursion, the less information we obtain, but the analysis is likely to be faster, as we have made the set of possible values smaller. We have chosen a compromise: if in a set of closures, one of the closures contain a set of closures that overlap the top-level set, then make the set recursive. The recursion is made by replacing the inner set of closures by an identifier, adding the extracted closures to the top level and making the projection recursive in the inserted variable.

8.4.3 Fixed-point iteration

Binding time analysis will be done by a fixed-point iteration using minimal function graphs [Jones *et al.* 86]. In this strategy we use a *minimal function graph* (MFG) which contain mappings of abstract functions to their result for specific argument sets (those that are actually needed). The initial MFG will contain only one mapping: the goal (abstract) function with its

parameters, binding the result to the top value (ID). Iteration proceeds by re-evaluating all mappings in the MFG. When a function call is needed, its value from the MFG is used. If there is no mapping for a call, one is added that maps the call to ID. If re-evaluation gives a result different from the one in the MFG, the mapping is changed to the greatest lower bound of the previous and the new value. As all changes are monotonic and there are only finitely many possible argument value combinations the iteration will terminate when a fixed-point is reached. This will then be a consistent solution, and it will be the largest possible as values start with the top element (ID) and are only made lower when necessary.

8.5 Examples

An example using polymorphism is shown below. First we define functions:

$$\alpha\ list\ =\ \mu\,\beta : nil\ void + cons\ \alpha \times \beta$$

$$f\ :\ (\alpha\ list) \rightarrow ((\alpha \times int) \times int\ list)$$

$$f\ x\ =\ m\,(m\ x)$$

$$m\ :\ (\alpha\ list) \rightarrow (\alpha \times int\ list)$$

$$\begin{aligned} m\ x\ =\ case\ x\ of\ \ & nil\ \ \ v : nil\ v\ ; \\ & cons\ v : cons\ ((fst\ v, 42), m\ (snd\ v)) \end{aligned}$$

Then the abstract functions:

$$f^{\#}\ :\ \mathcal{P}_{(\alpha\ list)} \rightarrow \mathcal{P}_{((\alpha \times int) \times int\ list)}$$

$$f^{\#}\ p_x\ =\ m^{\#}\ (m^{\#}\ p_x)$$

$$m^{\#}\ :\ \mathcal{P}_{(\alpha\ list)} \rightarrow \mathcal{P}_{(\alpha \times int\ list)}$$

$$\begin{aligned} m^{\#}\ p_x\ =\ case^{\#}\ p_x\ of\ \ & nil\ \ \ v : nil^{\#}\ v\ ; \\ & cons\ v : cons^{\#}\ \langle\langle fst^{\#}\ v, ID\rangle, m^{\#}\ (snd^{\#}\ v)\rangle \end{aligned}$$

Note that we have called $m^{\#}$ directly instead of using $apply^{\#}$ which we can do, as m is first order.

To find $f^{\#}\ fix\lambda g.ABSENT$ we construct an initial minimal function graph (MFG):

$$[(f^{\#}\ fix\lambda g.ABSENT\ \mapsto\ ID)]$$

Initially assuming the result is ID. During the fixed-point iteration we add function calls to the MFG, initially mapping them to ID. When we evaluate them we change their value to the new value. This yields the following sequence in the iteration:

$$[(f^{\#}\ fix\lambda g.ABSENT \mapsto ID)]$$

$$[(f^{\#}\ fix\lambda g.ABSENT \mapsto ID),\\ (m^{\#}\ fix\lambda g.ABSENT \mapsto ID)]$$

$$[(f^{\#}\ fix\lambda g.ABSENT \mapsto ID),\\ (m^{\#}\ fix\lambda g.ABSENT \mapsto ABSENT)]$$

$$[(f^{\#}\ fix\lambda g.ABSENT \mapsto ABSENT),\\ (m^{\#}\ fix\lambda g.ABSENT \mapsto ABSENT)]$$

$$[(f^{\#}\ fix\lambda g.ABSENT \mapsto ABSENT),\\ (m^{\#}\ fix\lambda g.ABSENT \mapsto ABSENT)]$$

As expected, changing the elements of a completely unknown list results in a completely unknown list. if we start with a list of unknown elements we get the following sequence:

$$[(f^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto ID)]$$

$$[(f^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto ID),\\ (m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto ID)]$$

$$[(f^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto ID),\\ (m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto\\ \qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle ABSENT, ID\rangle, g\rangle)]$$

$$[(f^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto ID),\\ (m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto\\ \qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle ABSENT, ID\rangle, g\rangle),\\ (m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle\langle ABSENT, ID\rangle, g\rangle \mapsto ID)]$$

$$[(f^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto ID),\\ (m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto\\ \qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle ABSENT, ID\rangle, g\rangle),\\ (m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle\langle ABSENT, ID\rangle, g\rangle \mapsto\\ \qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle\langle ABSENT, ID\rangle, ID\rangle, g\rangle)]$$

$$[(f^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto\\ \qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle\langle ABSENT, ID\rangle, ID\rangle, g\rangle),\\ (m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto\\ \qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle ABSENT, ID\rangle, g\rangle),\\ (m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle\langle ABSENT, ID\rangle, g\rangle \mapsto\\ \qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle\langle ABSENT, ID\rangle, ID\rangle, g\rangle)]$$

$$
\begin{aligned}
&[(f^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto \\
&\qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle\langle ABSENT, ID\rangle, ID\rangle, g\rangle), \\
&(m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle ABSENT, g\rangle \mapsto \\
&\qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle ABSENT, ID\rangle, g\rangle), \\
&(m^{\#}\ fix\lambda g.nil'\ ID \mp cons'\ \langle\langle ABSENT, ID\rangle, g\rangle \mapsto \\
&\qquad fix\lambda g.nil'\ ID \mp cons'\ \langle\langle\langle ABSENT, ID\rangle, ID\rangle, g\rangle)]
\end{aligned}
$$

Note that the two mappings for $m^{\#}$ are of different instances of the polymorphic type. There need not always be a mapping for every instance, as all mappings where the projection for the polymorphic part is ID or $ABSENT$ will be shared by all types.

Now we will show an example with higher order functions. In the example closures can be built to an arbitrary depth, which means that recursive function projections will be needed.

$$
\begin{aligned}
&tree\ =\ \mu\,\alpha : leaf\ int + node\ \alpha \times \alpha \\
&\\
&maxt\ :\ tree \rightarrow tree \\
&maxt\ t\ =\ max1\ (max2\ t) \\
&\\
&max1\ :\ (int \rightarrow tree) \times int\ \rightarrow\ tree \\
&max1\ t\ =\ (fst\ t)(snd\ t) \\
&\\
&max2\ :\ tree\ \rightarrow\ (int \rightarrow tree) \times int \\
&max2\ t\ =\ case\ t\ of\ \ leaf\ n : (f1, n); \\
&\qquad\qquad\qquad\qquad node\ p :\ max3\ (max2\ (fst\ p))\ (max2\ (snd\ p)) \\
&\\
&max3\ :\ (int \rightarrow tree) \times int\ \rightarrow\ (int \rightarrow tree) \times int\ \rightarrow\ (int \rightarrow tree) \times int \\
&max3\ t1\ t2\ =\ (f2\ (fst\ t1)\ (fst\ t2), max\ (snd\ t1, snd\ t2)) \\
&\\
&f1\ :\ int \rightarrow tree \\
&f1\ n\ =\ leaf\ n \\
&\\
&f2\ :\ (int \rightarrow tree)\ \rightarrow\ (int \rightarrow tree)\ \rightarrow\ int\ \rightarrow\ tree \\
&f2\ g1\ g2\ n\ =\ node\ (g1\ n, g2\ n)
\end{aligned}
$$

The function $maxt$ takes as argument a binary tree with integer leaves and returns a tree with the same structure but with all the leaves replaced by the maximum value of the leaves of the input tree. This is done by constructing a function that given an integer will build the tree with that integer and then supply this with the maximum value that was found in the same passage of the input tree as the function. Now follows the abstract functions (this time without type declarations):

$$max t^{\#}\ t\ =\ max1^{\#}\ (max2^{\#}\ t)$$

$$max1^{\#}\ t\ =\ apply^{\#}\ (fst^{\#}\ t)(snd^{\#}\ t)$$

$$\begin{aligned} max2^{\#}\ t\ =\ & case^{\#}\ t\ of \\ & \quad leaf\ n:\ \langle\{[f1]\}, n\rangle; \\ & \quad node\ p:\ max3^{\#}\ (max2^{\#}\ (fst^{\#}\ p))\ (max2^{\#}\ (snd^{\#}\ p)) \end{aligned}$$

$$max3^{\#}\ t1\ t2\ =\ \langle apply^{\#}\ \{[f2]\}\ (fst^{\#}\ t1)\ (fst^{\#}\ t2), (snd^{\#}\ t1)\ \sqcap\ (snd^{\#}\ t2)\rangle$$

$$f1^{\#}\ n\ =\ leaf^{\#}\ n$$

$$f2^{\#}\ g1\ g2\ n\ =\ node^{\#}\ \langle apply^{\#}\ g1\ n, apply^{\#}\ g2\ n\rangle$$

In the iteration below we start by giving $maxt$ a projection that keeps the structure of the tree but discards the leaves. We would thus expect a result projection of the same type. This projection has the form:

$$fix\lambda p_\alpha.leaf'\ ABSENT \mp node'\ \langle p_\alpha, p_\alpha\rangle$$

and will be referred to as pp in the example. For simplicity unreachable configurations are removed as soon as they are no longer reachable.

$$[(maxt^{\#}\ pp\ \mapsto\ ID)]$$

$$\begin{aligned} & [(maxt^{\#}\ pp\ \mapsto\ ID), \\ & (max2^{\#}\ pp\ \mapsto\ ID)] \end{aligned}$$

$$\begin{aligned} & [(maxt^{\#}\ pp\ \mapsto\ ID), \\ & (max2^{\#}\ pp\ \mapsto\ \langle\{[f1]\}, ABSENT\rangle), \\ & (max1^{\#}\ ID\ \mapsto\ ID), \\ & (max3^{\#}\ ID\ ID\ \mapsto\ ID)] \end{aligned}$$

$$\begin{aligned} & [(maxt^{\#}\ pp\ \mapsto\ ID), \\ & (max2^{\#}\ pp\ \mapsto\ \langle\{[f1]\}, ABSENT\rangle), \\ & (max1^{\#}\ \langle\{[f1]\}, ABSENT\rangle\ \mapsto\ ID), \\ & (max3^{\#}\ \langle\{[f1]\}, ABSENT\rangle\ \langle\{[f1]\}, ABSENT\rangle\ \mapsto\ ID)] \end{aligned}$$

$$\begin{aligned} & [(maxt^{\#}\ pp\ \mapsto\ ID), \\ & (max2^{\#}\ pp\ \mapsto\ \langle\{[f1]\}, ABSENT\rangle), \\ & (max1^{\#}\ \langle\{[f1]\}, ABSENT\rangle\ \mapsto\ ID), \\ & (max3^{\#}\ \langle\{[f1]\}, ABSENT\rangle\ \langle\{[f1]\}, ABSENT\rangle \\ & \qquad\qquad \mapsto\ \langle\{[f2\ \{[f1]\}\ \{[f1]\}]\}, ABSENT\rangle), \\ & (f1^{\#}\ ABSENT\ \mapsto\ ID)] \end{aligned}$$

$$[(maxt^{\#}\ pp \mapsto ID),$$
$$(max2^{\#}\ pp \mapsto \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle),$$
$$(max1^{\#}\ \langle\{[f1]\}, ABSENT\rangle \mapsto ID),$$
$$(max3^{\#}\ \langle\{[f1]\}, ABSENT\rangle\ \langle\{[f1]\}, ABSENT\rangle$$
$$\mapsto \langle\{[f2\ \{[f1]\}\ \{[f1]\}]\}, ABSENT\rangle),$$
$$(f1^{\#}\ ABSENT \mapsto pp)]$$

$$[(maxt^{\#}\ pp \mapsto ID),$$
$$(max2^{\#}\ pp \mapsto \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle),$$
$$(max1^{\#}\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle \mapsto ID),$$
$$(max3^{\#}\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle$$
$$\mapsto \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle),$$
$$(f1^{\#}\ ABSENT \mapsto pp),$$
$$(f2^{\#}\ \{[f1]\}\ \{[f1]\}\ ABSENT \mapsto ID)]$$

$$[(maxt^{\#}\ pp \mapsto ID),$$
$$(max2^{\#}\ pp \mapsto \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle),$$
$$(max1^{\#}\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle \mapsto ID),$$
$$(max3^{\#}\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle$$
$$\mapsto \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle),$$
$$(f1^{\#}\ ABSENT \mapsto pp),$$
$$(f2^{\#}\ fix\lambda c.\{[f1],[f2\ c\ c]\}\ fix\lambda c.\{[f1],[f2\ c\ c]\}\ ABSENT \mapsto ID)]$$

$$[(maxt^{\#}\ pp \mapsto ID),$$
$$(max2^{\#}\ pp \mapsto \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle),$$
$$(max1^{\#}\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle \mapsto ID),$$
$$(max3^{\#}\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle$$
$$\mapsto \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle),$$
$$(f1^{\#}\ ABSENT \mapsto pp),$$
$$(f2^{\#}\ fix\lambda c.\{[f1],[f2\ c\ c]\}\ fix\lambda c.\{[f1],[f2\ c\ c]\}\ ABSENT \mapsto pp)]$$

$$[(maxt^{\#}\ pp \mapsto pp),$$
$$(max2^{\#}\ pp \mapsto \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle),$$
$$(max1^{\#}\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle \mapsto ID),$$
$$(max3^{\#}\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle\ \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle$$
$$\mapsto \langle fix\lambda c.\{[f1],[f2\ c\ c]\}, ABSENT\rangle),$$
$$(f1^{\#}\ ABSENT \mapsto pp),$$
$$(f2^{\#}\ fix\lambda c.\{[f1],[f2\ c\ c]\}\ fix\lambda c.\{[f1],[f2\ c\ c]\}\ ABSENT \mapsto pp)]$$

This gave the expected result for the result of $maxt$. The interesting part is however the result of $max2$ which shows a recursive function projection. If we couldn't make recursive closures the iteration would yield larger and larger nonrecursive nested approximations.

8.6 Implementation Issues

In the rules for greatest lower bound and in other places we have either assumed that there are no name conflicts in the variables used to make recursive types and projections or else explicitly alpha-converted the names. In an implementation this can be be made simpler by using DeBruijn indexing of these variables. This also makes comparison of projections easier.

In some cases where higher order functions are used to build continuations the minimal function graph will contain separate mappings for all possible continuation structures (down to the depth where recursion is tied). As well as making the analysis slow, this also makes the result less useful, as we are really only interested in a single mapping using all possible continuations. A way of achieving this is to combine mappings in the MFG that have identical arguments except for the functional arguments, where the sets of closures are overlapping. The overlap requirement makes the chance of combining mappings that are really used in different contexts small. Note that the condition for combining mappings is the same as the condition for making a function projection recursive. Alternatively, a static analysis can find the set of closures that are possible values at a given program point, as sets of (function name, number of parameters) pairs. This can be used to determine when recursively nested closures are possible and when it is pertinent to combine mappings for a given function.

In the examples shown, the value of a mapping is always taken from the previous MFG, even if a new value has already been computed in the new MFG. Using the new value instead, in case it is already evaluated, will make the iteration shorter. If this strategy is used, re-evaluating the latest additions to the MFG first will improve iteration speed further.

8.7 Conclusion

We have presented an algorithm for binding time analysis of a higher order functional language with a polymorphic type system. It extends the ideas of [Launchbury 88] to higher order functions and polymorphic types, and compares well to [Nielson 88] by handling values that are *partially static*, containing both compile time and run time information in a single variable, which the present version of Nielson & Nielson cannot do (though I believe their basic framework could be extended to do so). By restricting the projection domains to contain only ID and $ABSENT$, a result similar to that of Nielson & Nielson can be obtained. The analysis is not as conceptually clean as Nielson & Nielson's, but we believe the increased precision is worth the cost.

Chapter 9

Miscellaneous Topics

This chapter contains a collection of ideas related to the subject of this thesis. Mainly they describe extensions to the methods shown in the previous chapters.

9.1 An optimization for grammar based binding time analysis

One of the reasons the grammar based BTA takes so long for large programs, is that there is a large number of variables that has the same binding time structure, and yet be represented by different non-terminals. This means that the right hand sides of many productions contain a large list of equivalent non-terminals, and this takes time to build. A simple example of this is the grammar:

$$\begin{aligned} x &\rightarrow S \mid P(D,x) \mid P(D,y) \mid P(D,z) \\ y &\rightarrow S \mid P(D,y) \mid P(D,x) \mid P(D,z) \\ z &\rightarrow S \mid P(D,x) \mid P(D,z) \mid P(D,y) \end{aligned}$$

which can be reduced significantly by combining non-terminals:

$$xyz \rightarrow S \mid P(D,xyz)$$

If this observation is to make any difference for the time taken to build the grammar, the non-terminals must be combined *before* BTA. This can be done by recognizing identifiers that will definitely have the same binding time, regardless of what that might be. Identifiers x and y will have the same binding time if x is *directly dependent* on y *and* vice versa. Direct dependence is defined by:

i) a formal parameter x to a function f is directly dependent on a variable y if y is used in a result position in an actual parameter to f in the position corresponding to x.

ii) a formal parameter x to a function f is directly dependent on a function g if a call to g is used in a result position in an actual parameter to f in the position corresponding to x.

iii) a function f is directly dependent on a variable y if y is used in a result position in the body of f.

iv) a function f is directly dependent on a function g if a call to g is used in a result position in the body of f.

v) a variable x in a *let*-binding is directly dependent on a variable y if the expression defining x contain y in a result position.

vi) a variable x in a *let*-binding is directly dependent on a function g if the expression defining x contain a call to g in a result position.

vii) an identifier a is directly dependent on an identifier c if there is an identifier b, such that a is directly dependent on b and b is directly dependent on c.

where we use "variable" both for formal parameters and variables in *let*-bindings. Identifiers also include function names. An expression e is in a result position in an expression d if:

d is e.

or d is $(IF\ d_1\ d_2\ d_3)$ and e is in a result position in d_2 or d_3.

or d is $(LET\ bindings\ d_1)$ if e is in a result position in d_1 and e is not shadowed by any of the bindings in *bindings*.

These rules can be used to group identifiers in equivalence classes of mutually directly dependent identifiers. Each of these equivalence classes are given a common non-terminal for use in BTA. If one of the classical efficient algorithms for finding strongly connected components in a graph is used, the construction of equivalence classes can be done relatively fast.

9.2 Extension of grammar based binding time analysis to higher order functions

This section describe a possible extension of the grammar based method for binding time analysis from chapter 6 and chapter 7 to higher order functions. We first sketch a value domain to use during partial evaluation and use this to construct an infinite binding time domain, which will be approximated by grammars as described in chapters 6 and 7.

9.2.1 The language

The language we will use is a curried form of the language used in section 3.1. A program is thus a set of untyped curried combinators using LISP primitives. The syntax is given below. We assume call-by-value, LISP-like semantics.

$$
\begin{array}{lll}
program & ::= & f_1\ x_{11} \ldots x_{1n_1} = exp_1 \\
 & & \ldots \\
 & & f_m\ x_{m1} \ldots x_{mn_m} = exp_m
\end{array}
$$

$$
\begin{array}{lll}
exp & ::= & x_{ij} \\
 & | & f_i \\
 & | & (QUOTE\ constant) \\
 & | & (CAR\ exp) \\
 & | & (CDR\ exp) \\
 & | & (ATOM\ exp) \\
 & | & (CONS\ exp\ exp) \\
 & | & (EQUAL\ exp\ exp) \\
 & | & (IF\ exp\ exp\ exp) \\
 & | & exp\ exp
\end{array}
$$

During normal evaluation we have the following value domain:

$$
\begin{array}{lll}
value & ::= & atom \\
 & | & (value\ .\ value) \\
 & | & [f_i\ value \ldots value]
\end{array}
$$

where $[f_i\ v_1 \ldots v_n]$ represents a closure of the function f_i with the partial list of arguments $v_1 \ldots v_n$. We assume that constants and input/output to the program are first-order values, that is they do not contain closures.

9.2.2 Partial evaluation

As in chapter 6 we will during partial evaluation use a tagged sum consisting of values (representing static values) and expressions (representing dynamic values) and partially static structures build from these. In addition to these we will also use closures with partially static arguments, yielding this partial evaluation domain:

$$
\begin{array}{lll}
p\_value & ::= & S(first\text{-}order\ value) \\
 & | & D(exp) \\
 & | & (p\_value\ .\ p\_value) \\
 & | & [f_i\ p\_value \ldots p\_value]
\end{array}
$$

Partial evaluation of expressions can then be done like this:

$$Peval[\![x_{ij}]\!]env = env\ x_{ij}$$

$$Peval[\![f_i]\!]env = [f_i]$$

$$Peval[\![(QUOTE\ c)]\!]env = S(c)$$

$Peval[\![(CAR\ e)]\!]env\ =$
  case $Peval[\![e)\ of$ $S((a\ .\ b))$: $S(a)$
  $D(e1)$: $D((CAR\ e1))$
  $(a\ .\ b)$: a
  otherwise : $error$
  end

$Peval[\![(CDR\ e)]\!]env\ =$
  case $Peval[\![e]\!]env\ of$ $S((a\ .\ b))$: $S(b)$
  $D(e1)$: $D((CDR\ e1))$
  $(a\ .\ b)$: b
  otherwise : $error$
  end

$Peval[\![(ATOM\ e)]\!]env\ =$
  case $Peval[\![e]\!]env\ of$ $S(a)$: $S(atom(a))$
  $D(e1)$: $D((ATOM\ e1))$
  otherwise : $S(false)$
  end

$Peval[\![(CONS\ e1\ e2)]\!]env\ =$
  case $(Peval[\![e1]\!]env, Peval[\![e2]\!]env)\ of$ $(S(a), S(b))$: $S((a\ .\ b))$
  (a,b) : $(a\ .\ b)$
  end

$Peval[\![(EQUAL\ e1\ e2)]\!]env\ =$
  case $(Peval[\![e1]\!]env, Peval[\![e2]\!]env)\ of$ $(S(a), S(b))$: $S(a\ =\ b)$
  $(a\ .\ b)$: $D((EQUAL\ dyn(a)\ dyn(b)))$
  end

$Peval[\![(IF\ e1\ e2\ e3)]\!]env\ =$
  case $Peval[\![e1]\!]env\ of$ $S(a)$: $if\ a\ then\ Peval[\![e2]\!]env\ else\ Peval[\![e3]\!]env$
  $D(e1)$: $D((IF\ e1\ dyn(Peval[\![e2]\!]env)\ dyn(Peval[\![e3]\!]env)))$
  $(a\ .\ b)$: $Peval[\![e2]\!]env$
  otherwise : $error$
  end

$Peval[\![e1\ e2]\!]env\ =$
  case $Peval[\![e1]\!]env\ of$ $[f_i\ v_1 \ldots v_k]$: $if\ n_i > k+1\ then$
  $[f_i\ v_1 \ldots v_k\ Peval[\![e2]\!]env]$
  $else\ if\ unfold?\ then$
  $Peval[\![exp_i]\!][\ {}^{x_{i1} \mapsto v_1, \ldots, x_{i(n_i-1)} \mapsto v_k,}_{x_{in_i} \mapsto Peval[\![e2]\!]env}\ ]$
  $else$ suspend call, details below
  $D(e1)$: $D(e1\ dyn(Peval[\![e2]\!]env))$
  otherwise : $error$
  end

The function *dyn*(...) converts a (partially) static value to a dynamic value by converting it to an expression. This might require suspension of closures constructed from incomplete applications. When suspending a closure or a call, the static parts of the parameters are added to the function name, yielding the name of the residual function. The dynamic parts become the parameters to the residual call. Details can be found in chapter 6.

When a residual expression has been generated, it is searched for suspended calls and closures. These are added to the list of functions that are needed in the residual program, and definitions for these will be generated later. Suspended closures are filled with dynamic parameters up to the full number of parameters to the function.

9.2.3 Binding time analysis

Binding time analysis will (as seen in section 4.3) make it possible to avoid a lot of the tests on the form of the partial values that form the parameters to base operators. This is done by approximating the set of possible partial values in a static analysis done before partial evaluation. We construct first this domain of binding time values, each of which approximates a set of partial values:

$$\begin{aligned} b\_value &::= D \mid s\_value^* \\ \\ s\_value &::= S \\ &\quad \mid (b\_value\ .\ b\_value) \\ &\quad \mid [f_i\ b\_value \ldots b\_value] \end{aligned}$$

So a binding time value is either D, representing all sets of partial values containing dynamic values or it is a list of descriptors: S describing all sets of first order static values, ($b1$. $b2$) describing the set of pairs of elements from $b1$ and $b2$ and $[f_i\ b1 \ldots bn]$ describing the set of closures of f_i with parameters in $b1 \ldots bn$. Even if we assume the lists contain no repetitions, these values can be arbitrarily large, giving an infinite binding time domain. However, by using grammars as described in chapters 6 and 7, we can generate finite recursive descriptions approximating these binding time values. In this way the list of *s_values* in a *b_value* will refer to non-terminals instead of to other *b_values*. In closures it is natural to use non-terminals associated with the parameters to the function in question, whereas it may be necessary to associate parameters of *CONS*-expressions with non-terminals to have appropriate non-terminals for the pairs in the list of *s_values*. Having a finite number of non-terminals thus makes the set of possible lists of *s_values* finite (if we avoid repetition).

Discussions of how to use the result of the binding time analysis can be found in chapters 6 and 7, [Bondorf 89] and [Bondorf *et al.* 88].

9.2.4 Signatures as higher order functions

In [Bondorf 88], the free signature of a term-rewriting system was reported to give problems for compilation by partial evaluation and for self-application. This was due to the fact that, although the language allowed arbitrarily many different constructors, each program would have a fixed finite set of constructors. So, a self-interpreter would have to simulate an infinite set of constructors using a finite set. This required a self-interpreter to use coded forms of data-structures, so any residual form of it would also use coding. This made it impossible to make compilation by partial evaluation of a self-interpreter yield satisfactory results: any object program would add a level of coding to the source programs data. Similar problems occured

when self-applying the partial evaluator. These problems made Bondorf drop the free signature in favor of a LISP-like data-structure in [Bondorf 89].

When having higher order functions it is possible to simulate a signature by a set of functions. Consider for example the signature

```
list = nil
     | cons value list
```

and a function using it:

```
append a b = case a of
                 nil          : b
                 cons a1 a2   : cons a1 (append a2 b)
```

This can be converted to the following set of definitions:

```
append a b          = a b (append1 b)
append1 b a1 a2     = cons a1 (append a2 b)
nil f g             = f
cons v l f g        = g v l
```

where the definitions of "nil" and "cons" corresponds to the declaration of the sort "list". This method is easily applicable to any signature, and is indeed used in some implementations of lazy functional languages [Peyton Jones *et al.* 89]. You will still have a finite set of "constructors" for each program, but now a residual program can have a different and larger set than the original! This is because, being functions, the constructors can be specialized. A self-interpreter need not code data-structures; if strong typing is not required, functional values can be represented by functional values with the same i/o properties. Indeed, it is possible to define a function that given a definition of a function returns the defined (curryed) function. As the type of the result of such a function is dependent on the value of the argument, this can of course not be done in any of the traditional strongly typed functional languages (ML, HOPE, Miranda *etc.*).

9.3 Retyping

To achieve variable splitting in the MIX-system, we propose to do *arity raising* using Sergei Romanenko's ideas from [Romanenko 88], where it is called *arity raising*.

The retyping consists of two phases: a type inference and a transformation.

The type inference will find types of variables as fixed tuple structures with either nil or general structures at the leaves.

The transformation transforms the program by splitting each tuple variable into a list of variables - one for each non-nil leaf of the tuple, and then change all function calls and bodies to confer to the changed parameter lists.

9.3.1 Type inference

The type inference uses a simple domain of types:

$$\begin{aligned}\langle Type\rangle ::= \; & nil \\ | \; & any \\ | \; & (\langle Type\rangle \,.\, \langle Type\rangle) \\ | \; & \bot\end{aligned}$$

nil means that the value is always nil. *any* is the top element in the domain and means that the value can be any structure. $(t_1 \,.\, t_2)$ means that the value always is a pair of two values of type t_1 and t_2. $\bot$ is the bottom element and is used as initial value in the analysis and as error value.

The values are reflexively partially ordered as follows

$$\begin{array}{rcll} \bot & \sqsubseteq & t & \text{for all } t \\ t & \sqsubseteq & any & \text{for all } t \\ (t_1 \,.\, t_2) & \sqsubseteq & (t_3 \,.\, t_4) & \text{iff } t_1 \sqsubseteq t_3 \text{ and } t_2 \sqsubseteq t_4 \end{array}$$

the operator $\sqcup$ is used for least upper bound in the type domain.

Given an environment ρ binding variables to types, the function T computes the type of an an expression.

$$\begin{array}{lcl}
T[\![variable\ v]\!]\rho & = & \rho(v) \\
T[\![(quote\ S-expr)]\!]\rho & = & C[\![S-expr]\!] \\
T[\![(car\ e)]\!]\rho & = & \begin{array}{ll} any & \text{if } T[\![e]\!]\rho = any \\ t_1 & \text{if } T[\![e]\!]\rho = (t_1 \,.\, t_2) \\ \bot & \text{otherwise} \end{array} \\
T[\![(cdr\ e)]\!]\rho & = & \begin{array}{ll} any & \text{if } T[\![e]\!]\rho = any \\ t_2 & \text{if } T[\![e]\!]\rho = (t_1 \,.\, t_2) \\ \bot & \text{otherwise} \end{array} \\
T[\![(atom\ e)]\!]\rho & = & any \\
T[\![(cons\ e_1\ e_2)]\!]\rho & = & (T[\![e_1]\!]\rho \,.\, T[\![e_2]\!]\rho) \\
T[\![(equal\ e_1\ e_2)]\!]\rho & = & any \\
T[\![(if\ e_1\ e_2\ e_3)]\!]\rho & = & T[\![e_2]\!]\rho \sqcup T[\![e_3]\!]\rho \\
T[\![(call\ e_1\ \ldots\ e_n)]\!]\rho & = & any \\
 & & \\
C[\![nil]\!] & = & nil \\
C[\![a]\!] & = & any \text{ if } a \text{ is an atom} \\
C[\![(c_1 \,.\, c_2)]\!] & = & (C[\![c_1]\!] \,.\, C[\![c_2]\!])
\end{array}$$

Type inference is done as a fixed-point iteration similar to the binding time analysis, using information about the types of the parameters of the goal function.

The reason for the type of a function call being *any* is that we do not attempt to split functions, but only parameters. An environment giving the types of functions could be added to the analysis if splitting of functions is wanted (as is perfectly possible using similar strategies).

9.3.2 Transformation: splitting of variables

Using the result of the type inference, variables with types different from *any* will be split. It is assumed that no type contains $\perp$, as this would imply that the program always would fail, or that a function would never be called.

A variable is split into a list of variables, one for each *any* leaf in the type of the variable. A variable x of type $(any\ any)$ (short for $(any.(any.nil))$) would thus be split into x_1 and x_2. Now a *building expression*, building the value of the old variable from the values of the new variables and *defining expressions*, defining the new variables in terms of the old are constructed. In the above example the building expression is $x = (cons\ x_1\ (cons\ x_2\ (quote\ nil)))$, and the defining expressions are $x_1 = (car\ x)$ and $x_2 = (car\ (cdr\ x))$.

In the body expression of the function of which x is a parameter all occurrences of x is replaced by the building expression. In all calls to the function (from anywhere in the program) the argument corresponding to x is replaced by a list of arguments constructed by substituting x by the original argument into the defining expressions of x_i, and using these new expressions as arguments. If (using the above example) the argument corresponding to x in a call is y, that argument is replaced by the argument list $((car\ y)\ (car\ (cdr\ y)))$. Then reduction is done using rules like

$$
\begin{aligned}
&(CAR\ (CONS\ a\ b)) = a \\
&(CDR\ (CONS\ a\ b)) = b \\
&(CAR\ (IF\ a\ b\ c)) \quad = (IF\ a\ (CAR\ b)\ (CAR\ c)) \\
&etc.
\end{aligned}
$$

The reduction rules and the type inference are dual in the sense that the reduction rules ensures that all the constructors that was introduced in the building expressions on the basis of the type inference will be eliminated by selectors from the defining expressions.

As an example consider the residual function

```
(Append1 (v) =
   (if (equal (car v) 'nil) then (car (cdr v))
   else (cons (car (car v))
              (call Append1
                   (cons (cdr (car v)) (cons (car (cdr v)) 'nil))
) )   )     )
```

and assume that the type of v is $(any\ any)$. Then the transformed version (before reduction) is

```
(Append1 (v1 v2) =
   (if (equal (car (cons v1 (cons v2 'nil))) 'nil)
```

```
   then (car (cdr (cons v1 (cons v2 'nil))))
   else (cons (car (car (cons v1 (cons v2 'nil))))
              (call Append1
                   (car (cons (cdr (car (cons v1 (cons v2 'nil))))
                              (cons (car (cdr (cons v1 (cons v2 'nil)))) 'nil)
                        )
                   )
                   (car (cdr (cons (cdr (car (cons v1 (cons v2 'nil))))
                                   (cons (car (cdr (cons v1 (cons v2 'nil)))) 'nil)
                   )   )   )
) )   )     )
```

and after reduction

```
(Append1 (v1 v2) =
   (if (equal v1 'nil) then v2
   else (cons (car v1) (call Append1 (cdr v1) v2))
) )
```

If an expression (if a (cons b (cons c) 'nil) (cons d (cons e 'nil))) is split into two expressions it could cause repeated calculation of a, *e.g.*, if it was split into (if a b d) and (if a c e). To avoid this we note that as we only split expressions inside function calls, we can simply move the conditional outside the call, transforming (call f ...(if a b c) ...) into (if a (call f ...b ...) (call f ...c ...)) before splitting the expressions in the argument list. This strategy might give some code duplication, but no repeated calculations, and assuming call-by-value, no unnecessary calculations either.

9.3.3 Results

The retyper has been implemented, and gives good results. With a self-interpreter it is now possible to make compilations where the number of parameters to functions in the object program is the same as in the source program, giving essentially an identity mapping.

Tests on compilers and cogen show that the parameter containing the dynamic argument values is split, but not the parameter containing the static values. This is because the static values are obtained from the list of needed functions, a variable the type inference can't find a composite type for, as the elements vary in length. The problem can be solved by sending the static values from through a "filter" controlled by the list of static variable names. This will give it a recognizable length, thus helping the type inferencer.

Below are tables of size and speed differences before and after retyping. The sizes are counted as number of cons-cells + number of non-nil atoms. The times are in seconds using le-Lisp on a SUN 3/50. After retyping, selfint.obj is as fast as selfint (but a little larger). The reason for the lack of size difference for cogen is due to the incomplete variable splitting in the "unfiltered"

version. cogen1 is cogen produced by a fsp with the above mentioned "filter". Neither cogen nor cogen1 show any speed improvement from variable splitting. This is propably because most of the work in cogen is done in functions that have no split parameters.

Since the call-graph analyzer (CGA) [Sestoft 88] unfolds calls only if no variable is used more than once, the retyper may, by splitting a variable into several, each of which is used only once, increase the number of calls that the CGA can unfold. By running the CGA after retyping (as well as before) the size of cogen1 can be further reduced to 8190, and the time for *L cogen1* ⟨*fsp1*⟩ to 13.9 s. Using the retyper yet another time yields no difference.

Program	Size before R.T.	Size after R.T.
selfint.object	1494	849
cogen	10976	10084
cogen1	11430	8994

Program	Time before R.T.	Time after R.T.	Time used for retyping
L selfint.object ⟨*input*⟩	33.6	21.7	0.5
L cogen ⟨*fsp*⟩	12.7	12.7	4.8
L cogen1 ⟨*fsp1*⟩	16.7	16.7	5.2

9.3.4 Retyping in CL

In [Mogensen 86] a language called CL was used. CL is similar to the above extended with a simple type system:

$$\begin{aligned}\langle type\rangle ::= \; & integer \\ | \; & real \\ | \; & string \\ | \; & structure\end{aligned}$$

where the *structure* type is similar to S-expressions, but with atoms being either *nil*, integers, reals or strings. CL contains for testing the type of atoms and for converting atoms to values of the corresponding type and vice versa. Note that an integer is not the same as an integer atom, the former is a machine integer, whereas the latter is a pointer to a record containing a type tag and a machine integer.

Retyping in CL consists of splitting structure typed variables into several new variables, some of which may have non-structure types, *or* changing the type of structure typed functions and variables into non-structure types, if their values always are atoms of a specific type.

The type domain used for the type inference is now:

$$\begin{aligned}\langle Type\rangle ::= \; & nil \\ | \; & integer\_atom \\ | \; & real\_atom \\ | \; & string\_atom \\ | \; & any \\ | \; & (\langle Type\rangle \,.\, \langle Type\rangle) \\ | \; & \bot\end{aligned}$$

Note that type inference is only done on structure typed variables, so no type inference domain is specified for the non-structure types. Type inference is done similarly to what was described above, with obvious extensions for operators that convert non-structure values to atoms. The Main difference is that, since we want to retype functions as well as variables, we can no longer just use *any* as the type of function results. Instead, we must use an environment of function types in addition to variable types. As we still don't want to split functions, we must convert any function result type of the form $(T1 \,.\, T2)$ into *any*.

As an example consider the residual function:

```
(v : structure) : structure =
   (IF (= (INTOF (CAR v)) 0)
       (CAR (CDR v))
       (CALL f (CONS (INTATOM (- (INTOF (CAR v)) 1))
                     (CONS (CAR (CDR (CDR v)))
                           (CONS (INTATOM (+ (INTOF (CAR (CDR v)))
                                             (INTOF (CAR (CDR (CDR v))))))
                           )       )
                           'nil
) ) )         )       )
```

The type inference will find:

v : $(int\_atom \,.\, (int\_atom \,.\, (int\_atom \,.\, nil)))$
f : int_atom

giving this transformed program:

```
f (v1 : int, v2 : int, v3 : int) : int =
   (IF (= v1 0)
       v2
       (CALL f (- v1 1) v3 (+ v2 v3))
   )
```

which is now recognizable as the fibonacci function.

The partial evaluator for CL was mainly used to specialize a ray-tracer program. For residual ray-tracer programs, the size and runtime was reduced by 25 - 30 percent (after the specialization reduced runtime by 80 percent).

9.3.5 General retyping

What we have seen are special cases of *retyping*; replacing a value by an equivalent value in another representation, modifying the program accordingly. In the retyper described above, particular instances of the general list type are replaced by fixed-length tuples, which are made part of the argument tuples of functions. In languages with richer type systems, more interesting transformations can be done, *e.g.*, creating new signatures from instances of a more general coding signature. This corresponds closely to specializing data structures, see section 9.2.4 for more ideas on this subject.

9.4 Dynamic Choice of Static Values

In all the previously described binding time analyses, the binding time of the result of a conditional with dynamic condition is dynamic, regardless of the binding times of the branches. This is a logical choice, as even if both branches contain static values, it will not be known at partial evaluation time which of these will be chosen. It is, however, possible to extend both the partial evaluator and the binding time analyzer to handle such values almost as static.

The main idea is to use the re-write rule:

$$(\mathsf{f}\ (\mathsf{if}\ \mathsf{a}\ \mathsf{b}\ \mathsf{c})) \Rightarrow (\mathsf{if}\ \mathsf{a}\ (\mathsf{f}\ \mathsf{b})\ (\mathsf{f}\ \mathsf{c}))$$

during partial evaluation to move a value consumer (f) into a place where the consumed value is static.

9.4.1 Extending the two-point domain

If we initially forget about partially static values, we can extend the two-point binding time domain with a third value C (for choice), obtaining:

$$S \sqsubseteq C \sqsubseteq D$$

where S and D represent the usual completely static and dynamic values, and C represent values that are a dynamic choice between static values. In other words, S, C and D represent sets of expressions obtainable at partial evaluation time:

$$\langle S \rangle ::= (\mathsf{quote}\ \langle \mathit{value} \rangle)$$

$$\begin{aligned} \langle C \rangle ::= & \langle S \rangle \\ & \mid\ (\mathsf{if}\ \langle D \rangle\ \langle C \rangle\ \langle C \rangle) \end{aligned}$$

$$\langle D \rangle ::= \langle \mathit{any\ expression} \rangle$$

An expression can represent a set of values: the set of values obtained by evaluating it with arbitrary values for the free variables in it. In this way S can be seen as the set of singleton sets of values, C as the set of finite sets of values and D as the set of all sets of values. This gives a more extensional view of the binding times, and makes it easier to justify some of the rules used in binding time analysis.

Binding time analysis must be extended to handle the extra C value. The propagation of binding times through expressions is shown below:

```
bta(exp,env) =
  case exp of
    name          : lookup(name,env)
    (quote v)     : S
    (car e)       : bta(e,env)
    (cons e1 e2)  : bta(e1,env) ⊔ bta(e2,env)
    ⋮             other base operators similarly
```

```
(if e1 e2 e3)     : case bta(e1,env) of
                      S : bta(e2,env) ⊔ bta(e3,env)
                      C : C ⊔ bta(e2,env) ⊔ bta(e3,env)
                      D : C ⊔ bta(e2,env) ⊔ bta(e3,env)
```

Function calls are handled as usual: update the parameters in the global environment and return the value of the function as found the global environment. The usual fixed-point iteration is used. There is, however, an interesting fact that we can use during binding time analysis: if a function has a C parameter, we know that we at partial evaluation time can move the conditionals in that parameter outside the call, leaving only static values in that parameter position. This allows us to convert the binding times of parameters from C to S when updating the global environment. No similar thing can be done for the result of a function.

This idea has been implemented (in 1985) as an extension to an early version of the MIX system, yielding results for compiler generation and self-application similar to those for the standard MIX. The main visible difference was that some programs could be non-trivially partially evaluated on the extended MIX, that couldn't be on the standard MIX. It was, however, normally possible to rewrite these programs to forms suitable for the unmodified MIX. Nothing was ever published about this system, and no later versions of MIX use the idea. Independently of this, Sergei Romanenko used almost exactly the same idea (down to calling the new binding time C) in an extension of the system described in [Romanenko 88]. As far as I know, no details of this has been published in the western world. My knowledge of it stems from personal discussions at the workshop of partial evaluation and mixed computation in October 1987.

9.4.2 Extension to partially static structures

The grammar based method for doing binding time analysis with partially static structures, can also be extended to handle binding times similar to the C above. When thinking of binding times as sets of sets of values, terminal symbols in the binding time grammars corresponds to specific sets of sets, and the grammars corresponds to recursive set equations. So, for example, the rule:

$$X \rightarrow atomS \mid (D\,.\,X)$$

correspond to the set equation:

$$X = atomS \cup \{(a\,.\,b) \mid a \in D,\ b \in X\}$$

where $atomS$ is the set of singleton sets of atoms and D is the set of all sets (of values, this will be assumed implicitly from now on). Thus "|" in the grammar rule is a union operator. It is generally used to combine the branches of a conditional with static test. We now introduce an operator "||", to be used when combining branches of a dynamic conditional. || is defined as the point-wise union operator:

$$A \,||\, B = \{a \cup b \mid a \in A,\ b \in B\}$$

This immediately gives us the rules:

$$
\begin{aligned}
A \,||\, D &= D \\
A \,||\, B &= B \,||\, A \\
(A \,||\, B) \,||\, C &= A \,||\, (B \,||\, C) \\
A \,||\, (B \mid C) &= (A \,||\, B) \mid (A \,||\, C)
\end{aligned}
$$

Note that while $A \mid A = A$, $A \,||\, A$ is generally *not* equal to A. The distributivity of $||$ over $\mid$ allows us to reduce any expression using these operators to a normal form. Using the usual techniques for building binding time grammars, we can build a grammar with these normal forms as right hand sides. Examples are:

$S \rightarrow atomS \mid (S \,.\, S)$	any static value
$X \rightarrow S \,||\, S$	either of two static values
$L \rightarrow atomS \,||\, (S \,.\, L)$	a list of known elements, but unknown length
$C \rightarrow C \,||\, C \mid S$	as the C value in the $S - C - D$ domain

Due to the large number of possible normal forms (given a reasonably sized set of non-terminals), it is likely that a fixed-point iteration will take very long time. It will therefore be a good idea to restrict the set in some way, *e.g.*, by limiting the number of $||$'s on a right hand side.

Chapter 10

Conclusion

We have in this thesis argued the necessity of explicit binding time information when partial evaluation is used for compiler generation, and we have presented a collection of algorithms for finding this binding time information. Furthermore, we have discussed how this information may be used to produce annotations describing different properties of parts of the program in question. In addition to actual binding time annotations, annotations can be used to guide unfolding of function calls and local bindings.

Chapters 6, 7 and 8 show rather different approaches. It is not intended that the latter chapters supercede the earlier, rather the methods have each their advantages and disadvantages, and in a given situation the method best suited for the problem must be used. An obvious example of this is the fact that the binding time analysis algorithms in chapter 6 and chapter 7 are suited for untyped languages, whereas the algorithm in chapter 8 is strongly tied to the type system of the language.

Most of the ideas presented in chapter 9 have not been implemented, but some of these will be natural to use for projects that are planned for the immediate future.

Bibliography

[Aho *et al.* 86] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers, Principles, Techniques, and Tools* Addison-Wesley 1986.

[Bondorf 88] A. Bondorf, *Towards a Self-Applicable Partial Evaluator for Term-Rewriting Systems* in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988.

[Bondorf 89] A. Bondorf, *A Self-Applicable Partial Evaluator for Term-Rewriting Systems* in the Proceedings of CCIPL'89, Springer Verlag, 1989.

[Bondorf *et al.* 88] A. Bondorf, N. D. Jones, T. Æ. Mogensen, P. Sestoft, *Self-application as a Tool for the Generation of Program Generators*, unpublished

[Bulyonkov 84] M. A. Bulyonkov. *Polyvariant Mixed Computation for Analyzer Programs*, Acta Informatica 21 pp.473-484, 1984.

[Bulyonkov 88] M. A. Bulyonkov. *A Theoretical Approach to Polyvariant Computation* in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988.

[Dybkjær 85] H. Dybkjær, *Parsers and Partial Evaluation*, student report, DIKU 1985.

[Ershov 82] A. P. Ershov. *Mixed Computation: Potential Applications and Problems for Study*, Theoretical Computer Science 18 pp.41-67, 1982.

[Futamura 71] Y. Futamura. *Partial Evaluation of Computation Processes - An Approach to a Compiler-compiler*, Systems, Computers, Controls 2(5) pp.721-728, 1971.

[Holst 88] N. C. K. Holst, *Language Triplets: the AMIX Approach*, in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988.

[Hughes 88] R. M. J. Hughes, *Backwards Analysis of Functional Programs*, in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988.

[Jones 86] N. D. Jones. *Flow Analysis of Lazy Higher Order Functional Programs*, in S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages,* Ellis Horwood, London, 1987.

[Jones 88] N. D. Jones, *Automatic Program Specialization: a Re-examination from Basic Principles* in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988.

[Jones *et al.* 86] N. D. Jones, A. Mycroft, *Data Flow Analysis of Applicative Programs Using Minimal Function Graphs* in the Proceedings of the Workshop on Programs as Data Objects, Copenhagen, October 1985, Springer Verlag LNCS 217, 1986.

[Jones *et al.* 85] N. D. Jones, P. Sestoft, H. Søndergaard, *An Experiment in Partial Evaluation: the Generation of a Compiler Generator* in Rewriting Techniques and Applications (ed. J.-P. Jouannaud), Dijon, France 1985, Springer Verlag LNCS 202, 1985.

[Jones *et al.* 88] N. D. Jones, P. Sestoft, H. Søndergaard, *Mix: a Self-Applicable Partial Evaluator for Experiments in Compiler Generation* in LISP and Symbolic Computation 1 3/4, 1988.

[Jørring *et al.* 86] U. Jørring and W. L. Sherlis, *Compilers and Staging Transformations*, in the Proceedings of the Thirteenth ACM POPL Symp., St. Petersburg, Florida 1986, pp. 86-96.

[Launchbury 88] J. Launchbury. *Projections for specialisation*, in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988.

[Marvell] A. Marvell, *To His Coy Mistress*

[Mogensen 86] T. Æ. Mogensen, *The Application of Partial Evaluation to ray-tracing*, masters thesis, DIKU 1986.

[Mogensen 88] T. Æ. Mogensen, *Partially Static Structures in a Self-Applicable Partial Evaluator* in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988. See also chapter 6 of this thesis.

[Mogensen 89b] T. Æ. Mogensen, *Binding Time Analysis for Polymorphically Typed Higher Order Languages* in the Proceedings of CCIPL'89, Springer Verlag, 1989. See also chapter 8 of this thesis.

[Mycroft 80] A. Mycroft. *The Theory and Practice of Transforming Call-by-Need into Call-by-Value*, Springer Verlag LNCS 83, 1980.

[Nielson 86] H. R. Nielson, F. Nielson, *Semantics Directed Compiling for Functional Languages* in the Proceedings of the ACM Conference on LISP and Functional Programming 1986.

[Nielson 88] H.R.Nielson, F.Nielson, *Automatic Binding Time Analysis for a Typed λ-Calculus* in Science of Computer Programming 10, North-Holland 1988.

[Peyton Jones *et al.* 89] S. L Peyton Jones, J. Salkind *The Spineless Tagless G-Machine* in the Proceedings of the 1988 Glasgow Workshop on Functional Programming.

[Reynolds 69] J. C. Reynolds. *Automatic Computation of Data Set Definitions*, Information Processing 68 pp.456-461, 1969.

[Romanenko 88] S. A. Romanenko, *A Compiler Generator Produced by a Self-Applicable Specializer can have a Surprisingly Natural and Understandable Structure*, in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988.

[Schmidt 88] D. A. Schmidt, *Static Properties of Partial Reduction* in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988.

[Sestoft 86] P. Sestoft. *The Structure of an Self-applicable Partial Evaluator*, in H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985,* pages 236-256, Springer Verlag LNCS 217, 1986.

[Sestoft 88] P. Sestoft, *Automatic call Unfolding in a Self-Applicable Partial Evaluator*, in the Proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark, October 1987, eds. D.Bjørner, A.P.Ershov and N.D.Jones, North-Holland 1988.

[Turchin 79] V. F. Turchin, *A Supercompiler System Based on the Language Refal*, SIGPLAN Notices 14(2) pp. 46-54, 1979.

[Turchin 82] V. F. Turchin, *Experiments with a Supercompiler*, in 1982 ACM Symposium on LISP and Functional Programming, pp. 47-55, ACM, 1982.

[Turchin 86a] V. F. Turchin, *Program Transformation by Supercompilation*, in H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985,* pages 236-256, Springer Verlag LNCS 217, 1986.

[Turchin 86b] V. F. Turchin, *The Concept of a Supercompiler*, ACM transactions on Programming Languages and Systems, 8(3) pp. 292-325, ACM 1986.

[Wadler 88] P. Wadler, *Deforestation: Transforming Programs to Eliminate Trees*, in the proceedings of ESOP'88, Springer Verlag LNCS 300, 1988.

[Wadler *et al.* 87] P. Wadler, R. M. J. Hughes *Projections for Strictness Analysis* in the Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference, Springer Verlag LNCS 274, September 1987.