Partial Evaluation using Rewrite Rules
A Specification of a Partial Evaluator for Similix in Stratego

Lennart Swart

13th September 2002

Abstract

Partial evaluation, or program specialization, is used when a variable is known to often hold one particular value. In that case, it is possible to specialize a program for that value, so the program performs better when the variable indeed has that value, as computations depending only on that variable(s) are already done.

Rewrite rules provide a good mechanism for expressing program transformations. As partial evaluation is an example of a program transformation, rewrite rules should provide a good mechanism for partial evaluation. Similix is a self-applicable partial evaluator for a large higher order subset of the strict functional language Scheme, a lisp-like language. Stratego is a programming language that provides us with rewrite rules which can be combined using rewriting strategies. An approach to an implementation of the process of partial evaluation as it is done by Similix 5.0, using the rewrite rules provided by Stratego is given.

The rewrite rules and strategies provided by Stratego prove to be a nice way to express partial evaluation. All phases a program has to pass to attain a partial evaluated version of the program can be split up in simple rewriting steps, which are combined in a neatly arranged manner by the rewriting strategies. The Stratego program stays clearly structured, as no complicated rules have to be used to do partial evaluation, and the strategies used to combine the rules can be set up in a logical way.

Contents

Chapter 1

Introduction

Partial evaluation, or program specialization, is used when a variable is known to often hold one particular value. In that case, it is possible to specialize a program for that value, so the program performs better when the variable indeed has that value, as computations depending only on that variable(s) are already done.

Rewrite rules provide a good mechanism for expressing program transformations. As partial evaluation is an example of a program transformation, rewrite rules should provide a good mechanism for partial evaluation. Similix is a self-applicable partial evaluator for a large higher order subset of the strict functional language Scheme, a lisp-like language. Stratego is a programming language that provides us with rewrite rules which can be combined using rewriting strategies. An approach to an implementation of the process of partial evaluation as it is done by Similix 5.0, using the rewrite rules provided by Stratego is given.

In this document an approach to a partial evaluator using rewrite rules and strategies is given. A program has to pass several phases before a partial evaluated version of the program is achieved. In the first chapters it will be made clear which phases a program has to pass, and in the last chapters those phases will be expressed in Stratego.

In Chapter 2 an introductory explanation of partial evaluation will be given. The subset of Scheme handled by Similix will be explained in Chapter 3. All information needed to start doing partial evaluation is then available, so in Chapter 4 the structure of the program used to do partial evaluation on the subset of Scheme will be explained.

Partial evaluation using rewrite rules is a process split up in four phases:

1. Desugaring, as described in Chapter 5
2. Binding time analysis, as described in Chapter 6
3. Doing some binding time improvements. These are described in Chapter 7
4. Partial evaluation, using the binding time analysis is described in Chapter 8

Based on these four steps, a program is developed that partially evaluates programs written in the large subset of Scheme. This program is written is Stratego using rewrite rules and strategies to combine those rules.

The reader is supposed to be familiar with the programming language Stratego, and to have some experience in programming in a functional language. In the rest of this paper, the language treated by the partial evaluator Similix is called Similix as well. To prevent confusion, a Similix procedure is referred to by procedure or function, a Stratego rule is called a rule.

Chapter 2

An introduction to partial evaluation

In this chapter an introduction to partial evaluation will be given. How partial evaluation of Similix using Stratego is done is not explained in this chapter, but in Chapters 6, 7 and 8. The basics of partial evaluation should be clear after this chapter. Not all rules for all constructs that can be come across in a program are given. Those rules are explained in the chapters where they are used (Chapters 6 and 8).

Partial evaluation is, as the word says, evaluating parts of a function. All parts of the program that can be evaluated will be evaluated. This means that:

- if no input to the function is known, all parts that can be evaluated in the original function will be evaluated.
- if some parts of the input to the function are known, all parts of the program that can be evaluated using that input are evaluated, and the rest is left as is.
- if the entire input to a function is known, all parts of the function are evaluated and partial evaluation becomes evaluation.

Partial evaluation can be used to specialize a function for a specific input, that is why partial evaluation is also known as specialization. For example, when specialization is done for the function:

```
(define (function a b) (+ a (* b b)))
```

for the arguments a = unknown and b = 2, the resulting function will be:

```
(define (function-2 a) (+ a 4))
```

as the result of (* b b) can be calculated if b is known. The name of the function is changed to indicate that the function is specialized for input 2.

Sometimes it is also possible to do partial evaluation without any known input. For example, when specialization is done for the function:

```
(define (perimeter r) (* (* 2 3.1415) r))
```

with the r being unknown, the resulting function will be:

```
(define (perimeter r) (* 6.2830 r))
```

as the result of (* 2 3.1415) can be calculated, even if r is unknown. The resulting function will be (a little) faster than the original one.

2.1 Online Partial Evaluation

Online partial evaluation is the most simple version of partial evaluation. First, all arguments to a function that are known are distributed through the program. For example, the function:

```
(define (function a b) (+ a (* b b)))
```

partially evaluated for a = unknown and b = 2, will look like this after distribution of the values known for the arguments:

```
(define (function a 2) (+ a (* 2 2)))
```

Then, everything that can be evaluated is evaluated:

```
(define (function-2 a) (+ a 4))
```

When another function call is encountered, this function is partially evaluated for its known input as well:

```
(define (function a b) (+ a (twice b)))

(define (twice a) (+ a a))
```

When the function is partially evaluated for a = unknown and b = 2, first the values known are distributed through the function:

```
(define (function a 2) (+ a (twice 2)))

(define (twice a) (+ a a))
```

Then it can be seen that another function call is encountered, so this function has to be partially evaluated as well:

```
(define (function a 2) (+ a (twice 2)))

(define (twice 2) (+ 2 2))
```

The distributed values can be used to do computation:

```
(define (function a 2) (+ a (twice-2)))

(define (twice-2) 4)
```

The result of the function twice-2 is a constant, so it can be inlined in the calling function:

```
(define (function-2 a) (+ a 4))

(define (twice-2) 4)
```

2.2 Offline Partial Evaluation

Offline partial evaluation is a process that takes two steps. In step one, the binding time analysis, all elements in the function are annotated as to whether or not they are reducible at partial evaluation time. Elements that are reducible are annotated as static, elements that are not reducible are annotated as dynamic. After binding time analysis is done, all elements annotated as reducible are reduced during partial evaluation. For example, the function:

```
(define (function a b) (+ a (* b b)))
```

partially evaluated for the input a = unknown and b = 2, gets annotated first. Note that a is a dynamic parameter and b is a static parameter. Static elements are annotated with a ":s" and dynamic elements with a ":d":

```
(define (function a:d b:s) (+ a:d (* b:s b:s)))
```

As both arguments to the "*" are static, its output is static as well. The second argument to the "+" becomes static, and the first argument is dynamic. As it is impossible to add an unknown value to a known value, the output to the "+" becomes dynamic:

```
(define (function a:d b:s) (+ a:d (* b:s b:s):s):d)
```

Partial evaluation can now be done for the static parts of the program, i.e., the multiplication:

```
(define (function-2 a) (+ a 4))
```

When another function call is come across during binding time analysis, the function called is analysed as well, just as partial evaluation was done as well whilst doing online partial evaluation:

```
(define (function a b) (+ a (twice b)))

(define (twice a) (+ a a))
```

These functions become, when analysed for a = unknown and b = 2:

```
(define (function a:d b:s) (+ a:d (twice b:s):s):d)

(define (twice a:s) (+ a:s a:s):s)
```

Because the call to twice with a static input results in a static value ((+ a:s a:s):s), the function call is annotated static as well.

One advantage of doing a binding time analysis first, and only then partially evaluate, is that binding time improvements can be done (see Chapter 7). For example:

```
(define (plus a b c d e f)
  (+ a b c d e f))
```

If a binding time analysis is done for the function shown above, and the function is called with input a = 1, b = 2, c = 3, d = 4, e = 5 and f = unknown, the analysed function looks like:

```
(define (plus a:s b:s c:s d:s e:s f:d)
  (+ a:s b:s c:s d:s e:s f:d):d)
```

Notice that the addition cannot be reduced, as one of the elements in the addition is dynamic. The result of partially evaluating this function is therefore:

```
(define (plus-1-2-3-4-5 f)
  (+ 1 2 3 4 5 f))
```

If a binding time improvement is done, that splits the arguments to an addition in two parts, the dynamic and static ones, and does two additions, one on all dynamic ones, with as last argument the addition of all static arguments, the binding time improved function looks like this:

```
(define (plus a:s b:s c:s d:s e:s f:d)
  (+ f:d (+ a:s b:s c:s d:s e:s):s):d)
```

Notice that the largest part of the addition can now be reduced, and after partial evaluation the function looks like this:

```
(define (plus-1-2-3-4-5 f) (+ f 15))
```

Another advantage of doing binding time analysis first, and only then partial evaluation, is that it is possible to reason about programs with partial input. When online partial evaluation is done for the function:

```
(define (onandon a b)
  (if (= a 0)
      a
      (onandon (- a 1) (+ b 1))))
```

with input a = unknown and b = 2, the result will be:

```
(define (onandon-2 a)
  (if (= a 0)
      a
      (onandon-3 (- a 1))))

(define (onandon-3 a)
  (if (= a 0)
      a
      (onandon-4 (- a 1))))

(define (onandon-4 a)
  (if (= a 0)
      a
      (onandon-5 (- a 1))))

etc.
```

Online partial evaluation for this function will never end, as a new function is created in every step for b being one more than in the previous step.

When offline partial evaluation is done, the function will look like this after the binding time analysis:

```
(define (onandon a:d b:s)
  (if (= a:d 0:s):d
      a:d
      (onandon (- a:d 1:s):d (+ b:s 1:s):s):d):d)
```

The conditional element in the if-expression is dynamic, as it can not be decided if a dynamic value is equal to zero. For a dynamic conditional element, no choice between the two branches can be made, so the entire if-expression is dynamic.
For completeness: the then-part of the if-expression is dynamic, because a is dynamic. The else-part of the if-expression is dynamic, because it has just been established that the body (i.e., the entire if-expression), of the function “onandon” with its first argument being dynamic ((- a:d 1:s):d) and its second element being static ((+ b:s 1:s):s) is dynamic.

When there is no possibility to get a static value for the only choice made before the next call to the recursive function, the partial evaluator can see that there is no possibility to end the recursion, given the now known input, and will not go on and on with partial evaluation, but will make just one partially evaluated function:

```
(define (onandon a b)
  (if (= a 0)
      a
      (onandon (- a 1) (+ b 1))))

(define (onandon-2 a)
  (if (= a 0)
      a
      (onandon (- a 1) 3)))
```

Chapter 3

Similix Language

Before a description of the partial evaluation of Similix is given the features of Similix are described. First, in Section 3.1, the core language is explained. Then, in Section 3.3 the rest of the language is explained. The abstract syntax given with each construction makes clear what the abstract syntax is, it is not a complete definition. The complete abstract syntax is given at the end of this chapter in Similix.r.

3.1 Core Language

Similix has five basic types. In the abstract syntax, the parser always parses the contents of those basic types as strings.

- Booleans: True, which is written as #t, and False: #f.
  The abstract syntax used for representing those values is: **Bool(**"#t"**)** and **Bool(**"#f"**)**.
- Characters: Single characters between single quotes.
  Abstract syntax: **Char(**"'a'"**)**. Because the contents of a basic type is always a string, the single quotes appear between double quotes.
- Numbers: Both integers and reals are supported.
  Abstract syntax: **Num(**"1"**)**, **Num(**"1.0"**)**.
- Strings: Strings between double quotes.
  Abstract syntax: **Str(**""abcde""**)**. Because the contents of a basic type is always a string, and a Similix string has double quotes as well, two pairs of double quotes appear.
- Symbols: Any string, without spaces, tabs, etc, without any quotation.
  Abstract syntax: **Sym(**"abcde"**)**.

Also, there are two ways to combine those types:

- Lists:
  - Pairs: For example: (1 . 2). Abstract syntax: **Lis2(** elem1, elem2 **)**.
  - Lists: For example: (1 2 3). Abstract syntax: **Lis1(** list of elements **)**.
- Vectors: For example: #(1 2 'a' 'b' "cd"). The abstract syntax is: **Vec(** list of elements **)**.

Further on, there are several ways to combine expressions into more complicated expressions:

- sequence-expressions: (begin exp1 exp2 ...). Abstract syntax: **Seq(** list of expressions **)**.
- lambda-expressions: (lambda (a b) (+ a b)). Abstract syntax: **Lambda(** list of variablenames, body **)**.
- an application: (function arg1 arg2). Abstract syntax: **App(** expression to be applied, list of arguments **)**.
- an if-then-else-expression: (if a b c). Abstract syntax: **If(** conditional expression, then-expression, else-expression **)**.
- a let-expression: (let ((c a) (d b)) (+ c d)). Abstract syntax: **Let(** list of bindings, body **)**, where a binding is a tuple of the variable name and the expression it is bound to. This is a parallel let-expression, meaning that all expressions to which the variables are bound are evaluated before the variables are bound.

Its also possible to load other similix files into a similix program, or to load some user defined constructors or primitives into a similix program. The similix files are actually included in the program, but the user defined constructors and primitives are not.

- load "file": loads a standard Similix-file. The contents of this file can be included in the file from which it is loaded.
- loads "file": loads a standard Similix-file. The contents of this file can be included in the file from which is loaded. The difference with the "load"-form is that all casematch- and caseconstr-expressions are are expanded into simpler forms that do not use pattern matching. The casematch- and caseconstr-expressions are explained in Section 3.3, where the sugar is explained. As desugaring is always done before partial evaluation, it doesn't matter to us if a load or a loads is used.
- loadt "file": loads a file in which user defined constructors or primitives are defined (.adt files). The structure of those files is explained in the next section (Section 3.2).

3.2 The structure of .adt-files

The structure of .adt-files is best illustrated through examples.

In .adt files user-defined constructors can be defined:

```
(defconstr (makepair fst *))
```

The constructor defined above defines a pair, the first element of that pair can be selected with the selector "fst". For selecting the second element of that pair no selector name is given, so a selector name is automatically created. The selectors are simply created using the name of the constructor, followed by a ".", followed by the index of the to be selected element. For the given example the second element of a pair is selected with the selector "makepair.1".

In an .adt file user-defined operators can be defined, which behave as if they were primitive. There are four ways to do this:

- (defprim (my-op x y) (cons x (cons x y)))
  A primitive named my-op is created which gets 2 arguments and does some concatenations. Abstract syntax:
  KOVE1(Key, Operator-name, Variables, scheme-Expression**)**.
- (defprim 1 my-sqrt sqrt)
  A primitive named my-sqrt is created which gets 1 argument (the 1 is the arity), and returns the square root of that argument.
  Abstract syntax: **KAOV(**Key, Arity, Operator-name, scheme-Variable**)**.
- (defprim plus +)
  A primitive named plus is created which gets a variable amount of arguments and returns the addition of those arguments.
  Abstract syntax: **KOV(**Key, Operator-name, scheme-Variable**)**.
- (defprim (plus . x) x)
  A primitive named plus is created, which gets a variable amount of arguments, and has to be defined elsewhere. This construction can be used to give a primitive different properties than it originally had. Abstract syntax:
  KOVE2(Key, Operator-name, Variable, scheme-Expression**)**.

The way to define properties of a primitive, is by giving them different keys. The key used in all examples given previously is “defprim”. When a primitive is defined with the key defprim, it means that the primitive can be reduced at partial evaluation time when all its arguments are static. There are some other keys, which give some other properties to primitives, though they can be treated in two different ways:

- defprim, defprim-transparent, defprim-tin.
  These keys declare a primitive that can be reduced when all its arguments are static.
  - defprim and defprim-transparent are exactly the same. Most primitives can be defined with a defprim-transparent, and therefore those are treated as “normal”, and can be defined with a defprim as well.
  - defprim-tin (transparent if needed). Primitives defined with this key are transparent as well, the difference with the normal defprim-form is that primitives defined with defprim-tin can be applied an infinite number of times (think of: +, it is always possible to add a number to an other number). Primitives defined with defprim or defprim-transparent sometimes can only be applied a specific number of times (think of: cdr, you can take the tail of a list, as long as there are elements left in the list, i.e., (cdr (cdr (cdr '(1 2)))) is not possible). Primitives defined with defprim-tin can be defined with defprim or defprim-transparent as well, not changing the behaviour of the primitive, but when defprim-tin is used some extra information about the primitive is added.
- defprim-dynamic, defprim-opaque, defprim-abort, defprim-abort-eoi.
  Primitives defined with these keys are never reduced, and left in the residual program as they were found regardless of the input to the primitives. (think of: read, it is impossible to read something from a file during partial evaluation time, as the filename may be static, but the contents of the file need not be static)
  - defprim-dynamic defines primitives that should never be reduced. The output of of applying a primitive defined with defprim-dynamic is always dynamic, regardless of the input.

  - defprim-opaque defines primitives that are evaluation order dependent. For example: read, when two occurences of a read operation are evaluated in the wrong order, they read each others input, and therefore do not function as they should anymore.
  - defprim-abort defines primitives that abort execution. Such a primitive is never reduced.
  - defprim-abort-eoi defines primitives that abort execution, but where the evaluation order does not matter. (eoi = evaluation order independent). For example, when errors about an input are to be reported to the user, it does not matter in which order these errors are given, as long as they are given

3.3 Sugar

All constructions mentioned in the following paragraphs can actually be rewritten to the constructions mentioned in Section 3.1. This process is called desugaring. How this is done is shown in Chapter 5. It is however useful to understand the different constructions before the desugaring is explained.

3.3.1 Conditional Expressions

Next to the standard if-then-else expression mentioned in Section 3.1 there are some other conditional expressions:

- An if-then-expression: (if a b), which is actually the same as the standard if-then-else expression with a dummy value inserted at the else-expression. The abstract syntax used for this expression is **IfThen(** a, b **)**.
- A conditional expression: (cond (condition1 then-part1) (condition2 then-part2) ...). The abstract syntax used for this expressions is **Cond(** list of tuples of conditions and then-parts **)**
- A conditional expression with an else-part: (cond (condition1 then-part1) (condition2 then-part2) ... (else else-part)). Note that the "else" can also be written as "_". Abstract syntax: **CondElse(** list of tuples of conditions and then-expressions, else-expression **)**

3.3.2 Let Expressions

Next to the standard parallel let-expression, for which the order in which the bindings are evaluated is unspecified, there are some other types of let-expressions:

- A sequential let-expression, in which the first binding is evaluated first, the second binding is evaluated second, etc. This means the following two expressions are actually the same:

  Sequential let:

  ```
  (let* ((a b)
         (c d)
         (e f))
        (+ a c e))
  ```

  Normal (parallel) let:

```
(let ((a b))
     (let ((c d))
          (let ((e f))
               (+ a c e))))
```

The abstract syntax used for representing sequential let-expressions is: **LetStar(** list of bindings, body **)**, where a binding is a tuple of the variable name and the expression it is bound to.

- A named let-expression, which can be used for defining local procedures with initial values. For example the following piece of code defines a loop which prints the numbers from one to nine:

  ```
  (let loop ((i 0))
       (display i)
       (if (< i 10)
           (loop (+ i 1))))
  ```

  The loop starts with variable i being 0, and gets called with i increased at every loop. Abstract syntax: **LetP(** name of the named let, list of bindings, body **)**.

- A recursive let-expression. In a normal let-expression the bindings of the variables are only known inside the body of this let-expression. Sometimes we want to define local recursive functions, i.e., local functions that can be called from within its definition. Using a normal let this is not possible, as within the definition of a binding the binding itself is not known. If we want to define a function that prints the numbers from one to nine using only normal recursive functions (i.e., no named lets, the way we did it in the previous example), it will look like this:

  ```
  (letrec
     ((loop (lambda (a)
              (display a)
              (if (i < 10)
                  (loop (+ i 1))))))
     (loop 0))
  ```

  The abstract syntax used for representing recursive let-expressions is: **LetRec)** list of functions defined, body of the let-expression **)**, where a function is a three-tuple of the name of the function, the arguments to that function, and the body of that function.

3.3.3 Boolean Operators

There are some boolean operations:

- And: (and a b ...). Abstract syntax: **And(** list of expressions **)**.
- Or: (or a b ...). Abstract syntax: **Or(** list of expressions **)**.

3.3.4 Case Expressions

There are two different kinds of case-expressions, which are not found in ordinary Scheme, but can be used in Similix:

- A casematch expression: This matching form is used for matching ordinary Scheme expressions constructed by cons.
- A caseconstr expression: This matching form is used for matching user defined constructors.

Similix.r

On the next two pages the signature of Similix is given.

```
module similix
signature
   constructors
    OCDs: List (OCD) -> OCDs
    Program: List (TLE) * List (D) * List (TLE) -> Program
    Loadt: F -> TLE
    Define: P * List (V) * B -> D
    Body: List (D) * List (E) -> B
    File: String -> F
    Var: String -> V
    Proc: String -> P
    Const: String -> C
    S: String -> S
    Num: Num -> SE
    Str: Str -> SE
    Char: Char -> SE
    Bool: Bool -> SE
    Sym: String -> Sym
    SE: SE -> K
    Quote: Dat -> K
    Dat1: SE -> Dat
    Dat2: Sym -> Dat
    Dat3: Lis -> Dat
    Dat4: Vec -> Dat
    Lis1: List(Dat) -> Lis
    Lis2: List(Dat) * Dat -> Lis
    Vec: Dat -> Vec
    K: K -> E
    V: V -> E
    ConsQ: C -> E
    P: P -> E
    C: C -> E
    S2: S -> E
    Seq: List(E) -> E
    Lambda: List(V) * B -> E
    App: E * List(E) -> E
    If: E * E * E -> E
    Let: List(Tuple([V,E])) * B -> E
    Ofa: String -> Ofa
    OfaS: OfaS -> Ofa
    OfaS2: String -> OfaS
    OfaE: Ofa -> E
    Ova: String -> Ova
    OvaS: OvaS -> Ova
    OvaS2: String -> OvaS
    OvaE: Ova * List(E) -> E
    Key: String -> Key
    Ari: String -> Key
    Ss: String -> Ss
    KOVE1: Key * Ofa * List(V) * SchE -> OCD
    KOVE2: Key * Ova * V * SchE -> OCD
    KAOV: Key * Ari * Ofa * SchV -> OCD
    KOV: Key * Ova * SchV -> OCD
    Defconstr: List(Tuple([C,List(Ss)])) -> OCS
```

```
 Loadt2: F -> OCD
 Ss2: S -> Ss
 E: E -> SchE
 V4: V -> SchV


 Load: F -> TLE
 Loads: F -> TLE
 IfThen: E * E -> E
 Cond: List(Tuple([E,List(E)])) -> E
 CondElse: List(Tuple([E,List(E)])) * List(E) -> E
 LetStar: List(Tuple([V,E])) * B -> E
 LetP: P * List(Tuple([V,E])) * B -> E
 LetRec: List(Tuple([P,List(V),B])) * B -> E
 And: List(E) -> E
 Or: List(E) -> E
 EmptyPat: String -> MPat
 ElseWic: String -> WiC
 CaseMatch: E * List( Tuple([MPat,List(E)]) ) -> E
 CaseConstr: E * List( Tuple([CPat,List(E)]) ) -> E
 DatE: Dat * List(E) -> MPatA
 MPatB: MPat * List(E) -> MPatA
 MPatA: MPatA -> MPat
 K2: K -> MPat
 MPat2: MPat * MPat -> MPat
 V2: V -> MPat
 WiC: WiC -> MPat
 CPat: C * List(CPat) -> CPat
 V3: V -> CPat
 WiC2: WiC -> CPat


Name : String * String -> Name
```

Chapter 4

Architecture

Partial evaluation is done by running four programs consecutively on the input. The input is a normal similix program, with the exception that it contains one extra procedure. This procedure has to have the name "bta", and no arguments. The body of this function contains one application, applying a function defined elsewhere in the program to a set of arguments. These arguments can either be constants or variables. Partial evaluation is done on the procedure mentioned in the body of the function "bta", where the procedure is partially evaluated to all constants in its arguments. All variables are treated as dynamic input, and the name is ignored. It is therefore allowed to use the same variable twice. For example:

```
(define (bta) (function dynamic 1 2 dynamic some-other-variable))
```

starts partial evaluation for the function "function" with its second and third arguments being 1 and 2 respectively, and all other arguments being dynamic.

Partial evaluation is done in four phases:

1. The first step towards partial evaluation is desugaring the program, i.e., rewriting the program with sugar (see Section 3.3) to the core-language as described in Section 3.1. This process, which does not yet require the "bta" function as it does nothing with input yet, is described in Chapter 5.

2. After the input-program has been desugared, binding time analysis is done on the program to determine which parts of the program can be reduced, and which parts of the program stay as they are, depending on the input. This binding time analysis is described in Chapter 6.

3. Binding time improvements can be done now, see Chapter 7.

4. Now that the static and dynamic parts of the program are known, the parts annotated static can be evaluated. How this is done is described in Chapter 8.

The programs all import language signatures, files in which the abstract syntax used in the Stratego programs are defined. They also use some auxiliary programs, mostly programs that are used in two files. The program Similix-Desugar loads the VarRenamer, a program that renames bound variables that are bound elsewhere in the function as well. Which language signatures are imported by which programs, and which auxiliary programs are loaded, can bee seen in figure 4.1.

The different programs used for these phases, and how the Similix programs pass through these programs, can be seen in figure 4.2. The different extensions used for the different files can also be seen in this figure. As can be seen from this picture, it is also possible to just desugar a program, or just desugar and do binding time analysis on a program. The pp used in figure 4.2 actually consists of two parts, as can be seen in figure 4.3. The first part, Explicit-Annotations, makes all Stratego-annotations explicit, so they can be pretty-printed by the pretty-printer.

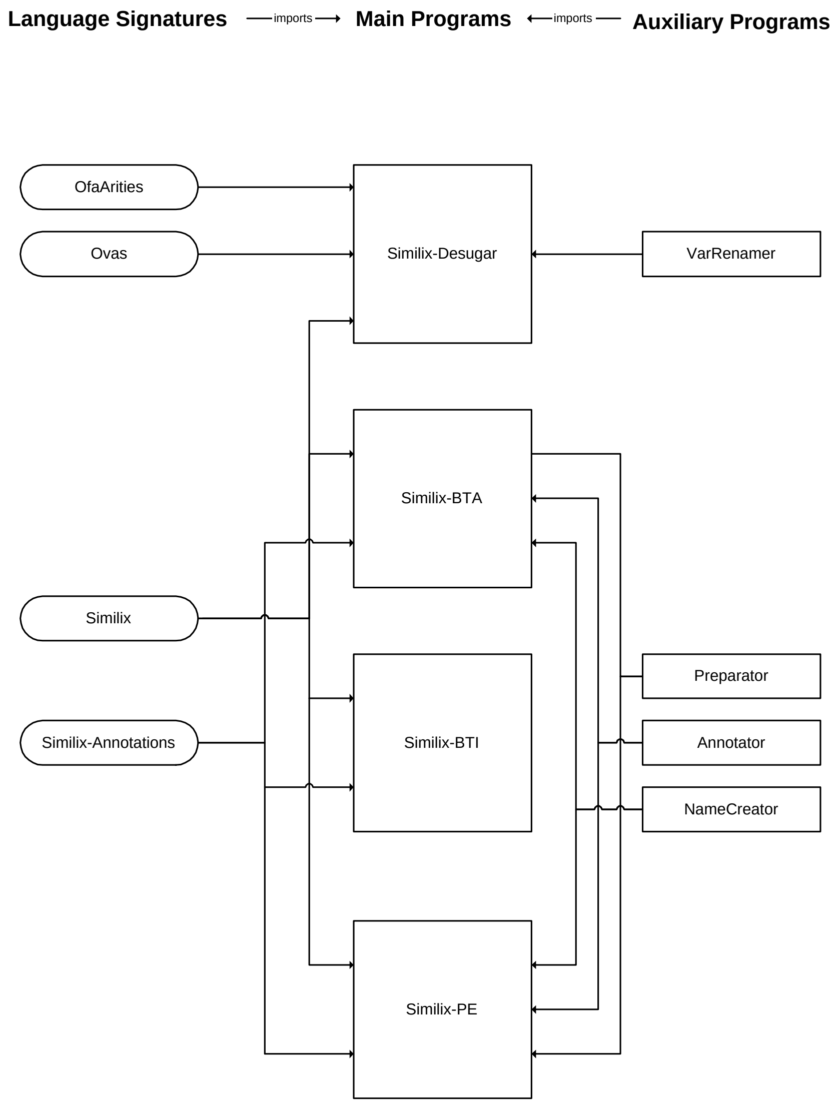

Figure 4.1: The files used in the program. This figure shows which files are imported in the main programs

When describing partial evaluation, names are created for the binding time analysed versions and the partially evaluated versions of the functions. These names contain all information about the input given for a function. The names and their meanings are best described through some examples.

The function having the name function-a-s-d-s-<ssd>-<[sss]>, is a function which has been annotated, i.e., on which binding time analysis has been done. This can be seen by the "-a", for annotated. The first element passed to the func-

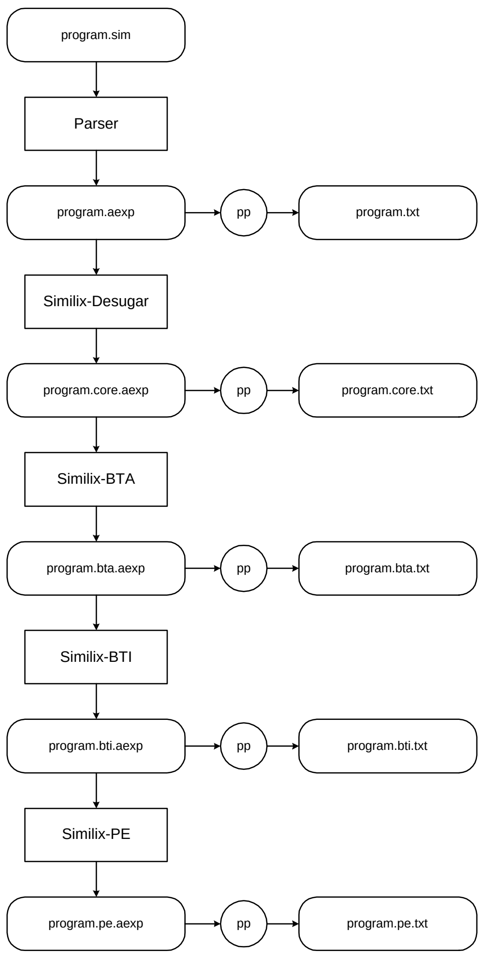

Figure 4.2: An illustration of how the input flows through the program. The pp is a pretty-printer that consists of multiple pieces. This pp is shown in figure 4.3.

tion was static ("-s"), the second dynamic ("-d"), the third static again ("-s"). The fourth element is placed between < and >, meaning the argument has more than one annotation. All annotations between only < and > are annotations for constructors. In this case the first "s" says the constructor that is applied is known, the second element that the first element in the constructor is static, and the third ,says that the second element in the constructor is dynamic. The input could have been (my-constructor 1 a). The fifth element is described by its annotations between "<[" and "]>", meaning the input was a list, with three static elements. Combinations of those signs are possible as well, when the input for a function "example" is, for example, (my-constructor '(1 2) (my-constructor 3 4 a) '((1 2) (3 4))), the annotated version of the example function would be:

```
example-a-<s<[ss]><sssd><[<[ss]><[ss]>]>>
```

The names of partially evaluated functions are constructed in almost the same way as are the names of binding time analysed functions. The only difference is that some extra "-"s are added, because else a list containing the integers 2 and 3 would look like: <[23]>, which can be a list containing 23, as well. Therefore a list containing 2 and 3 looks like : <[-2-3]>, and a list containing 23 like: <[-

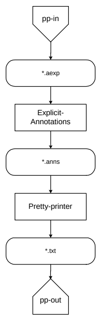

Figure 4.3: The programs used to pretty-print a program.

23]>, a list containg -23 looks like: <[--23]>. Variables are notated as a “-_”. So, when the function “function” with input: 2 a ’a’ (my-constructor 1 a) ’(1 2 3), has to be given a name, it would look like: function-pe-2-_-’a’-<-1-_>-<[-1-2-3]>. The partially evaluated version of the function “example” as used before (input: (my-constructor ’(1 2) (my-constructor 3 4 a) ’((1 2) (3 4)))), becomes:

```
example-pe-<-<[-1-2]>-<-3-4-_>-<[-<[-1-2]>-<[-3-4]>]>>
```

4.1 An example

To illustrate the architecture of partial evaluation using rewrite rules, an example will be given in this section. The example used here is the ackermann-function:

```
(define (ackermann m n)
  (if (= m 0)
      (+ n 1)
      (if (= n 0)
          (ackermann (- m 1) 1)
          (ackermann (- m 1) (ackermann m (- n 1))))))

(define (bta) (ackermann 2 n))
```

This function does not do anything interesting when used, but gives interesting results when partially evaluated. When desugaring is done on this program (gmake ackermann.core.txt), the result will be the same as the original program, as no sugar is used in this program.

When binding time analysis is done (gmake ackermann.bta.txt), the result will be:

```
(define (ackermann m n)
  (if (= m 0)
      (+ n 1)
      (if (= n 0)
          (ackermann (- m 1) 1)
          (ackermann (- m 1) (ackermann m (- n 1))))))
```

```
(define (ackermann-a-s-s m:S n:S)
  (if (= m:S 0:S):S
      (+ n:S 1:S):S
      (if (= n:S 0:S):S
          (ackermann-a-s-s
              (- m:S 1:S):S
              1:S):S
          (ackermann-a-s-s
              (- m:S 1:S):S
              (ackermann-a-s-s
                  m:S
                  (- n:S 1:S):S):S):S):S):S)

(define (ackermann-a-s-d m:S n:D)
  (if (= m:S 0:S):S
      (+ n:D 1:S):D
      (if (= n:D 0:S):D
          (ackermann-a-s-s
              (- m:S 1:S):S
              1:S):S
          (ackermann-a-s-d
              (- m:S 1:S):S
              (ackermann-a-s-d
                  m:S
                  (- n:D 1:S):D):D):D):D):D)

(define (bta)
  (ackermann 2 n))
```

The original function is always included in the program, and all binding time analysed versions of this function needed for partial evaluation are added. If both arguments to the function are static, the whole function, and all constructs in it are static. If only the first argument is static, some parts of the program are static, and other parts are dynamic. Note that the conditional expression (= m 0) is static, and (= n 0) is dynamic. Because of this, this second conditional expression will be present in the partially evaluated program (gmake ackermann.pe.txt):

```
(define (ackermann m n)
  (if (= m 0)
      (+ n 1)
      (if (= n 0)
          (ackermann (- m 1) 1)
          (ackermann (- m 1) (ackermann m (- n 1))))))

(define (ackermann-pe-1-1)
  3)

(define (ackermann-pe-0-2)
  3)

(define (ackermann-pe-1-0)
  2)

(define (ackermann-pe-0-1)
  2)

(define (ackermann-pe-2-_ n)
  (if (= n 0)
      3
      (ackermann-pe-1-_ (ackermann-pe-2-_ (- n 1)))))

(define (ackermann-pe-1-_ n)
  (if (= n 0)
      2
      (ackermann-pe-0-_ (ackermann-pe-1-_ (- n 1)))))

(define (ackermann-pe-0-_ n)
  (+ n 1))

(define (bta)
  (ackermann 2 n))
```

Note that some functions representing a value are present in the resulting program. The calls to these functions are replaced by those values, and therefore these functions are never called, and can be ignored.

Chapter 5

Desugaring

Before binding time analysis is started some desugaring is done. This simplifies binding time analysis and partial evaluation. In addition to the real desugaring, some other things are done to simplify future binding time analysis and partial evaluation.

```
desugar =
  loadfiles
; bottomup(try( NamedLetToLetRec
              + InternalDefsToLetRec
              + ListToSeq))
; renamevars
; topdown(try(lambdalift))
; Program(id,map(LetRecToSepFunc);concat,id)
; simplerewritings
; bottomup(try(MoveLambda))
; lookuparis
; eta
; Program(id,map(MkName),id)

simplerewritings =
  topdown(try( CondToIfThen
             + CondElseToIf
             + IfThenToIf
             + AndToIf
             + OrToIf
             + LetExpand
             + SeqExpand
             + (  CaseMatchToCond1
               <+ CaseMatchToCond2)
             + (  CaseConstrToCond1
               <+ CaseConstrToCond2)))
```

5.1 Inclusion of loaded files into the program

Files loaded using a load or loads expression have to be replaced by the text in the referenced file.

As included files can load files again, a while loop is used to load files. The difference between the load-form and the loads-form is that caseconstr- and casematch-expressions in files that are included using a loads statement are first rewritten into a simpler form, without caseconstr- and casematch-statements. Because this is always done during desugaring, there is no need to distinguish between the loads- and load-expressions. Therefore first all loads-expressions are rewritten to load-expressions (LoadsToLoad).

Then **Load(** file **)** is replaced by **Load(** contents **)** where contents is the contents of the file (LoadLoad).

As a last step all functions defined in those contents are copied into the program, as are all load- and loads-expressions in those contents (IncludeLoad). All load- and loads-expressions just copied into the program are handled in the next round of the while-loop.

```
strategies
  loadfiles =
    while(FilesToBeLoaded,
            ( Program( map(try(LoadsToLoad);try(LoadLoad) )
                     , id
                     , map(try(LoadsToLoad);try(LoadLoad) ))
            ; IncludeLoad))
  ; repeat(LoadtFile)

rules
  FilesToBeLoaded =
    ?Program(a,_,c);
      where(<conc
            ; filter(Load(id) +
                     Loads(id))>(a,c) => [_|_])

  LoadsToLoad :
    Loads(filename) ->
    Load(filename)

  LoadLoad =
    Load(ParseSimilix)

  IncludeLoad :
    Program(a,b,c) ->
    Program(<concat>[listofloadts,a2,c2],<conc>(b,b2),[])
      where <conc
            ; filter(Load(id))>(a,c) => listofloads;
            <conc
            ; filter(Loadt(id))>(a,c) => listofloadts;
            <map(?Load(Program(<id>,_,_)))
            ; concat>listofloads => a2;
            <map(?Load(Program(_,_,<id>)))
            ; concat>listofloads => c2;
            <map(?Load(Program(_,<id>,_)))
            ; concat>listofloads => b2
```

5.2 Parsing Similix files

Before Similix files can be included in the program, they have to be parsed to be understood by Stratego. The parsing is done through a call to the program sglr, which parses the file and puts the result of parsing in a new file. After that, implode-asfix is called on this new file, resulting in a .aexp file, which can be read by Stratego. After this .aexp file is read, both new files are deleted.

```
  ParseSimilix :
    filename -> contents
    where new-file => newfile1
        ; new-file => newfile2
        ; <parse-similix>(<un-double-quote>filename
                         , newfile1)
        ; <call>("implode-asfix", [ "-i", newfile1
                                  , "-o", newfile2])
        ; <ReadFromFile> newfile2 => contents
        ; <call>("rm",["-f",newfile1,newfile2])

  parse-similix =
    ?(file1, file2);
    <call>("sglr", ["-2","-p","Similix.tbl"
                   ,"-i",file1
                   ,"-o",file2])
```

5.3 Removal of internal definitions

A body having internal definitions is rewritten to a recursive let.

Internal definitions, which are function declarations inside a body of functions that are only accessible inside that body, are the same as functions declared in a recursive let-expression with as body the body for which the internal definitions are defined.
For example:

```
(define (add a b)
  (define (plus c d) (+ c d))
  (plus a b))
```

is the same as:

```
(define (add a b)
  (letrec ((plus (lambda (c d) (+ c d))))
    (plus a b)))
```

Stratego code:

```
  InternalDefsToLetRec :
    Body([def|defs], exps) ->
    Body( []
        , [ LetRec(<map(\Define(a,b,c) -> (a,b,c) \ )>[def|defs]
          , Body([],exps))])
```

5.4 Removal of multiple expressions in a body

A body which consists of more than one expression is expanded into begin-form.

For example:

```
(define (plus a b)
  (display "adding")
  (+ a b))
```

is rewritten to:

```
(define (plus a b)
  (begin (display "adding")
         (+ a b)))
```

Stratego code:

```
  ListToSeq :
    Body([],[a,b | rest]) ->
    Body([],[Seq([a,b | rest])])
```

5.5 Rewriting named let-expressions to recursive let-expressions

Named (possibly recursive) let-expressions are rewritten into recursive let-expressions. Remember that the following two expressions represent the same:

```
(let loop ((i 0))
     (display i)
     (if (< i 10)
         (loop (+ i 1)))))

(letrec
   ((loop (lambda (a)
             (display a)
             (if (i < 10)
                 (loop (+ i 1))))))
   (loop 0))
```

Stratego code:

```
  NamedLetToLetRec :
    LetP(procname,varsvals,body) ->
    LetRec([ (procname,vars,body)]
           , Body([],[App(V(procname),vals)]))
      where <map(?(<id>,_))>varsvals => vars ;
            <map(?(_,<id>))>varsvals => vals
```

5.6 Bound variable renaming

Bound variables are renamed, if they are already bound elsewhere. This simplifies the rest of the desugaring, and later binding time analysis and partial evaluation. At this moment in desugaring, the only places where variables can be bound are definitions, let-expressions, lambda-expressions and recursive let-expressions.

Bound variable are renamed in one topdown traversal. For each newly introduced variable, it is checked if a renaming rule already exists for this variable. If this is the case, new rules are created which rename the variable to a new variable. These rules are applied in the rest of the topdown traversal. When the topdown traversal is finished, all variables are bound only once.

When a function definition is met in the traversal, rules are created which rename the variables bound by the definition to themself, i.e., do nothing. From now on, renaming rules for this variable exists, so if this variable is bound somewhere else in the program, the variable will be renamed to a new variable.

When a let-expression is come across, the renaming rules are applied to the right hand sides of the bindings, as these do not fall in the scope of those bindings. If there are any variables in the left side of the bindings which are already bound, i.e., for which rename rules already exist, new rules are created which rename those variables to new variables. For all other variables, rules are created which rename the variable to itself.

When a lambda-expression is come across, all arguments to that lambda-expression which are already bound, are renamed and rules are created to do this renaming in the rest of the program. Rules that do nothing are created for all other arguments.

When a recursive let-expression is encountered, all its elements are temporarily put in a RecLet-constructor, to limit the scope of a function defined in a LetRec to that function. All new arguments to a RecLet-expression are renamed to themselves, all arguments already bound outside the RecLet-expression are renamed and rules are created to do this renaming in the future.

```
module VarRenamer
imports Similix TemporaryAnnotations lib dynamic-rules

strategies
  renamevars =
    rec r( try(Let(map((id,r)),id))
         ; try(LetRec(map(MkRecLet),id))
         ; {| RenameVar,
              RenameStr :
              try( RenameVars
                 + RenameVar
                 + RenameStr
                 )
            ; (Let(map((r,id)),r) <+ all(r))
           |})
rules

  RenameVars =
    ?Let(bindings,body)
  ; <map(RenameBindingVars)>bindings

  RenameBindingVars =
    ?(x,y)
  ; <RenameVar>V(x)
  ; new => z
  ; <CreateRules>(x,z)

  RenameBindingVars =
    ?(x,y)
  ; not(<RenameVar>V(x))
  ; <CreateRules>(x,x)

  RenameVars =
    ?Define(name,args,body)
  ; <zip(CreateRules)>(args,args)

  RenameVars =
    ?Lambda(names,body)
  ; <map(try(VarRenamer))>names
```

```
MkRecLet :
  (a,b,c) -> RecLet(a,b,c)

RenameVars :
 RecLet(procname,args,body) ->
 (procname,args,body)
   where <map(try(VarRenamer))>args

VarRenamer =
  ?x
; <RenameVar>V(x)
; new => z
; <CreateRules>(x,z)

VarRenamer =
  ?x
; not(<RenameVar>V(x))
; <CreateRules>(x,x)

CreateRules =
  ?(a,b)
; rules(RenameVar : V(a) -> V(b)
        RenameStr : a -> b)
```

5.7 Simplifying let-expressions

Parallel and sequential let-expressions are expanded into nested simple let-expressions, each having only one binding.

The difference between a sequential let- and a normal parallel let-expression is that the order in which the bindings are evaluated is not defined for a parallel let, but is defined for a sequential let expression. As the bindings of a parallel let have to be evaluated in some order, it is chosen to evaluate them in sequential order, too. Therefore let-expressions and let*-expressions can be treated in the same way. Stratego code:

```
LetExpand :
  Let([(a,b),(c,d) | rest],e) ->
  Let([(a,b)],Body([],[Let([(c,d) | rest],e)]))

LetExpand :
  LetStar([(a,b),(c,d) | rest],e) ->
  Let([(a,b)],Body([],[Let([(c,d) | rest],e)]))
```

5.8 Symplifying sequences

Sequences are rewritten to sequences with only two expressions in each begin-form. Stratego code:

```
SeqExpand :
  Seq([a,b,c | rest]) ->
  Seq([a,Seq([b,c | rest])])
```

5.9 Removal of pattern matching

When a casematch (or caseconstr, which is rewritten in exactly the same way, but will not be explained here) is rewritten, it is first necessary to bind the to be matched expression to a variable, because the to be matched expression can have side-effects, and can therefore only be evaluated once. This, of course, is not necessary when only one case is represented with the casematch-expression, because then it is possible to rewrite the casematch to an if-then-else-expression using the to be matched expression only once (CaseMatchToCond4).

To rewrite a casematch-expression having more than one case, the root of the casematch-expression is first looked up and the to be matched expression is bound to a new variable. In the same rewriting step, the first match is rewritten to an ordinary if-then-else-expression using the new variable in the conditional-expression. The rest of the matches are rewritten to a casematch-expression which matches the new variable to the expressions (CaseMatchToCond5).

This casematch-expression is then rewritten in a topdown traversal. When, in the topdown traversel, an if-then-else-expression using a variable in the conditional-expression, with a casematch expression trying to match that same variable in the else-expression is come across, the first match of the casematch is rewritten to a if-then-else-expression in a very simple way (CaseMatchToCond3). When such an if-then-else-expression with a casematch-expression with only one case is come across, this case is rewritten to an if-then-expression (CaseMatchToCond2). When an if-then-else-expression with a casematch trying to match against a WiC-expression (i.e., an else-expression), the contents of this WiC-expression is simply used as the else-branch to the if-then-else-expression (CaseMatchToCond1).
Stratego code:

```
strategies
  patternmatchtocond =
    topdown(try( ( CaseMatchToCond1
                 + CaseMatchToCond2
                 + CaseMatchToCond3)
              <+ ( CaseMatchToCond4
                 + CaseMatchToCond5)))

rules

  CaseMatchToCond1 :
    If( App(p,[var,q])
      , r
      , CaseMatch(var,[(WiC(a),b)])) ->
    If( App(p,[var,q])
      , r
      , b)

  CaseMatchToCond2 :
    If( App(p,[var,q])
      , r
      , CaseMatch(var,[(a,b)])) ->
    If( App(p,[var,q])
      , r
      , IfThen(App(Ofa("equal?"),[var,a])
      , b))
        where not(<?WiC(_)>a)
  CaseMatchToCond3 :
    If( App(p,[var,q])
      , r
      , CaseMatch(var,[(a,b),(c,d) | rest])) ->
    If( App(p,[var,q])
      , r
      , If( App(Ofa("equal?"),[var,a])
          , b
          , CaseMatch(var,[(c,d) | rest])))

  CaseMatchToCond4 :
    CaseMatch(var,[(a,b)]) ->
    IfThen( App(Ofa("equal?"),[var,a])
          , b)

  CaseMatchToCond5 :
    CaseMatch(var,[(a,b), (c,d) |rest]) ->
    Let( [(V(z),var)]
       , Body([],If( App(Ofa("equal?"),[V(z),a])
                   , b
                   , CaseMatch(V(z),[(c,d) | rest]))))
      where new => z
```

5.10 Removal of recursive let-expressions

Recursive let-expressions are rewritten into separate functions. Variables that are free in the bodies of the let-expressions are added as parameters (Lambdalifting).

The rewriting of recursive let-expressions to separate functions is done in two steps. First, the free variables from each function mentioned in the LetRec-expression are collected. Those free variables are then added as parameters to the function definition. Also, a dynamic rule which adds the collected free variables to the arguments of an application of this function, is created. In this way, all free variables in a function are passed to this function, and those free variables are accepted as parameters, and are not free anymore.

In step two, the functions mentioned in the LetRec-expression are rewritten to separate defines. The LetRec-expression is removed.

Example:

```
(define (function a b)
  (letrec ((addall (lambda (c) (+ a b c)))
           (addfst (lambda (c) (+ a c)))
   (addsnd (lambda (c) (+ b c))))
  (addsnd (addfst (addall a)))))
```

is rewritten to:

```
(define (addall c a b)
  (+ a b c))

(define (addfst c a)
  (+ a c))

(define (addsnd c b)
  (+ b c))

(define (function a b)
  (addsnd (addfst (addall a a b) a) b))
```

Note that the function addall had a and b as free variables. That are exactly the variables added as parameters in the desugared version. The function addfst only needs a, so a is added as an parameter to the new function-definition, and passed as an argument to the function as well. The function addsnd only needed an extra b, so b is added.
Stratego code:

```
strategies

strategy =
  topdown(try(lambdalift));
  Program(id,map(LetRecToSepFunc);concat,id);

  lambdalift =
    rec r({| AddArgs :
             try( LambdaLiftRuleGenerator +
                  AddArgs );
             all(r)
          |})

rules
  MkLambdaLiftRules :
    (procname, vars, body) ->
    (procname, <conc>(vars,fvs),body)
    where <freevars>Define(procname, vars, body) => fvs
        ; rules(
            AddArgs :
              App(V(procname),args) ->
              App(V(procname),<conc>(args,<map(!V(<id>))>fvs))
          )

  LetRecToSepFunc :
    Define(name,args,body) ->
    <conc>(extradefines,[Define(name,args,body2)])
      where < collect-all(
               ?LetRec(<map(\ (a,b,c) -> Define(a,b,c) \ )>,_))
            ; concat>body => extradefines;
            <topdown(
              try(?LetRec(_,Body([],[<id>])))
                   )>body => body2
```

5.11 Moving of lambda-expressions

Lambda-expressions are moved as far upwards in the program as possible, to bind the arguments in the lambda-expression directly. For example:

```
(define (func)
  (((lambda (a b) (+ a b)) 1) 2))
```

is rewritten to:

```
(define (func)
  ((lambda (b) ((lambda (a) (+ a b)) 1)) 2))
```

The a is bound to 1 directly, and the b to 2.

5.12 Looking up arities of applications

Lambda-expressions are added to all applications which require more arguments then they have. Before this can be done it is necessary to know how many arguments an application actually needs. First the arities for all built-in primitive functions are read in. The function OfaArities reads the file OfaArities.aexp, which holds tuples of the name of the fixed arity operator, and its arity. Rules can then be created to rewrite variables of procedure-names whose names equals the name of the fixed arity operator to Ofas holding that name, and the arity as a second element in a list.

Because all strings ending with a "?" are automatically parsed as constructor-tests without that questionmark, some extra work has to be done for primitives whose name end with a questionmark. All primitives whose name end with a questionmark (ASCII-number 63), are filtered from the input as read from the file OfaArities.aexp. Then a three-tuple is created, with the first element being the primitive's full name, the second element being the primitives name without the questionmark and the third element being the arity of the primitive. All constructor-tests whose name equals the second element of these three-tuples can now be rewritten to Ofas with as name the first element of the three-tuple (the name, including the questionmark), and with the arity as mentioned in the last element of the three-tuple.

Next, all operators with variable arity are read in from the file Ovas.aexp. All variables and procedure-names having that name will be rewritten to Ovas. No arity information is added, as operators with variable arity need no arity information. The reason this step is done in this phase of desugaring is that next to adding arity-information also the parser is corrected. The parser can, of course, not tell by a name if something is a fixed arity operator, a variable arity operator, a procedure name, a constructor name, etc. In this phase of desugaring this information is added to the program as well.

```
strategies
  lookuparis =
    OfaArities
  ; LookupOvas
  ; Program( map(where(try(UserdefinedConstrs)))
           , map(LookupDefines)
           , id)
  ; topdown(try( LookupAri
               + LookupOva1
               + LookupOva2
               + LookupAriCQ
               + LookupAriC
               + LookupAriS
               + LookupAriP))

rules
   OfaArities =
    where ( <ReadFromFile;?Aris(<id>)>
               "Ofas/OfaArities.aexp" => ofas
          ; map(\ Ari(a,b) -> (a,<string-to-int>b) \ )
          ; map(\ (a,b) -> (a,b)
              where rules( LookupAri  :
                             V(a) ->
                             Ofa([a,b])) \ )
          ; filter(where(?(<IsConstructorTest>,<id>)))
          ; map( \ (a,b) -> (<split-before;Fst>(a,"?"),b) \ )
          ; map( \ (a,b) -> (a,b)
              where rules( LookupAri :
                             ConsQ(a) ->
                             Ofa([<conc-strings>(a,"?"),b])) \ ))

  IsConstructorTest = explode-string;last;?63

  LookupOvas =
    where ( <ReadFromFile;?Ovas(<id>)> "Ovas/Ovas.aexp"
          ; map(\Ova(x) -> x
              where rules(LookupOva1 : V(x) -> Ova(x)
                          LookupOva2 : P(x) -> Ova(x)) \ ))
```

Next the user defined operators and constructors have to be annotated with their arities and be rewritten to be of the right kind. As all loadt-statements have been moved to the front of the program earlier, the function UserdefinedConstrs can be mapped over the first part of the program to handle all loadt-ed files. A loadt-ed file consists of an OCDs-constructor containing all operators and constructors defined in that file. Therefore the file loadt-ed is parsed, and the contents are put in the variable contents, which now is a list of all operators and constructors defined in that file. All KAOVs (Key, Arity, Operator-name, scheme-Variable) and KOVE1s (Key, Operator-name, Variables, scheme-Expression), the only ways to define a operator with fixed arity are filtered out of these contents, and rewritten to tuples containing only the name and arity of these operators. These tuples can now be used to define rewrite rules which rewrite variables of procedure names to Ofas with arity information.

Now the constructor definitions are filtered out of the contents of the file. The selector names are created from these constructors. For all selectors for which a name is given in the file, this name is used, and for all selectors for which no name but a “*” is given, the name is constructed. Selectors select elements from a constructor, so selectors always have arity one. Dynamic rules are created to rewrite all variables having the same name as these selectors to selectors, and to add the information that the arity is one. Variables having the same name as the constructors are rewritten to constructors. A constructor gets an arity equal to the number of selectors it has. A constructor-test gets arity information saying the arity of the test equals one.

Now all expressions defining an operator with variable arity are filtered out, and all variables having a name equal to that name are rewritten to Ovas, having no arity information.

```
rules
  UserdefinedConstrs =
    ?Loadt(filename);
      where ( <ParseSimilix;?OCDs(<id>)>filename => contents
            ; filter(KAOV(id,id,id,id) + KOVE1(id,id,id,id))
            ; map( (\KAOV(a,b,c,d) -> (c,<string-to-int>b) \ )
                 + (\KOVE1(a,b,c,d) -> (b,<length>c) \ ) )
```

```
                      => lookuplist

          ; list(MakeOfaRules)
          ; < filter(?Defconstr(<id>))
            ; concat> contents => temp
          ; map(?(_,<length;dec;upto <+ []>)) => indices
          ; < zip(MakeSelNames)
            ; concat
            ; list(MakeSelRules)>(<map(AddSelNames)>temp,indices)

          ; <map(\ (a,b) -> (a,<length>b) \ )>temp
          ; list(MakeDefconstrRules)
          ; list(MakeConstrTestRules))
          ; < filter( KOV(id,id,id)
                    + KOVE2(id,id,id,id))
            ; map(MkOvaRules)>contents

AddSelNames :
  (a,b) ->
  <map(\ belem -> (a,belem) \ )>b

MakeSelNames :
  (name,indices) ->
  <zip(try(ConcSelNames))>(name,indices)

ConcSelNames :
  ((name,"*"),index) ->
  <concat-strings>[name,".",<int-to-string>index]

ConcSelNames :
  ((name,Ss2(orig)),index) ->
  orig

MakeDefconstrRules :
  (name,ari) ->
  (name,ari)
    where rules(LookupAriC : V(name) -> C([name,ari]))

MakeConstrTestRules :
  (name,ari) ->
  (name,ari)
    where rules(LookupAriCQ : ConsQ(name) -> ConsQ([name,1]))

MakeSelRules :
  name ->
  name
    where rules(LookupAriS : V(name) -> S([name,1]))

MakeOfaRules :
  (name,ari) ->
  (name,ari)
    where rules(LookupAri : V(name) -> Ofa([name,ari]))

MkOvaRules =
  ?KOV(key,name,def)
  ; rules(LookupOva1 : V(name) -> Ova(name)
          LookupOva2 : P(name) -> Ova(name))

MkOvaRules =
  ?KOVE2(key,name,var,def)
  ; rules(LookupOva1 : V(name) -> Ova(name)
          LookupOva2 : P(name) -> Ova(name))
```

Information about the arity of procedure-calls is read from the definitions of those procedures, and is added to the procedure calls through a dynamic rule.

```
  LookupDefines =
    ?Define(a,b,c);
    where (rules(LookupAriP : V(a) -> P([a,<length>b])))
```

5.13 Eta-expansion

After all variables are rewritten using the dynamic rules just created eta-expansion can be done. If a procedure-call with arity 0 is encountered, then an application to an empty list is created. For all other applications of expressions for which eta-expansion is possible, there are two possibilities.

The first possibility is that eta-expansion is not necessary, i.e., the application is already an application to the number of arguments mentioned in the arity-information. In this case all that is done is removing the arity-information.

The second possibility is that eta-expansion is necessary. In this case a list of new variables is created, and the arguments to the application are enlarged with those new variables. The application is put inside a lambda-expression binding those new variables, and the arity-information is removed.

Finally, it is possible that a expression for which eta-expansion is possible is found by itself, i.e., not inside an application. If the arity information for such an expression reads 0, nothing is done. Else, an application is added, which applies the expression to a newly created list of arguments, which are bound by a newly created lambda-expression.

```
strategies
  eta =
    topdown( try(  EtaP
                <+ Eta
                <+ Eta2)
           ; try(RemoveEmptyLambda))

rules

  EtaP :
     P([name,0]) ->
     App(P(name),[])

  Eta :
    App(constr#([[name,ari]]),args) ->
    App(constr#([name]),args)
      where <Is-Eta>constr
          ; <eq>(ari,<length>args)
```

```
Eta :
  App(constr#([[name,ari]]),args) ->
  Lambda( newargs
        , Body( []
              , [App(constr#([name])
                    , <conc>(args,newargs1))]))
    where <Is-Eta>constr
        ; not(<eq>(ari,<length>args))
        ; < subt
          ; dec
          ; upto
          ; map(new)>(ari,<length>args) => newargs
        ; map(\ x -> V(x) \ ) => newargs1

Eta :
  Lambda( a
        , Body( []
              , [Lambda( b
                       ,Body( []
                            , [App(constr#([c]),d)]))])) ->
  Lambda( <conc>(a,b)
        , Body([]
              , [App(constr#([c]),d)]))
    where <Is-Eta>c

Eta2 :
  constr#([[name,0]]) ->
  constr#([name])
    where <Is-Eta>constr

Eta2 :
  constr#([[name,ari]]) ->
  Lambda( args
        , Body( []
              , [App(constr#([name]),args1)]))
    where <Is-Eta>constr
        ; <dec;upto;map(new)>ari => args
        ; map(\ x -> V(x) \ ) => args1

Is-Eta = ?"Ofa"
       + ?"C"
       + ?"ConsQ"
       + ?"S"
       + ?"P"
```

Chapter 6

Binding Time Analysis

Binding time analysis is split up in three phases. First of all it may be required to look up the body and names of the arguments of a function. A mechanism for this is needed. Then some rules have to be created for selectors. After those preparations are set up, the real binding time analysis can be performed. In Section 6.1 the preparations are explained, in Section 6.2 the strategy used to do binding time analysis is explained, and in Section 6.3 binding time analysis of a single function is explained.

6.1 Preparations

To do a binding time analysis it is necessary to be able to look up the functions defined in the similix file, when those functions are called from within a function being analysed. Secondly, its necessary to sort out which selectors select which element from a constructor.

6.1.1 Prepare functions to be looked up

To make sure it is possible to look up a function when only an application of that function is known, it a mechanism that will give the names of the arguments of that function and the body of that function given its name and arity is required. To achieve this, some dynamic rules are created before binding time analysis.

```
MakeFunctionsLookupable :
  Define(name,args,body) ->
  Define(name,args,body)
    where <length>args => number
        ; rules( LookupFunction:
                   (name,number) ->
                   (args,body) )
```

6.1.2 Preparation of selectors

When doing binding time analysis on selectors its necessary to know which selector selects which argument from a specific constructor. To be able to decide this during binding time analysis a set of rules needs to be constructed beforehand. Because the names of the selectors can be given by the Similix programmer, it is necessary to read the contents of a "loadt"-ed file and construct the right names for the constructors from those contents. First, using the function MkConstrSelNames all names are created as if no names were given in the .adt-file. For example, the selector names constructed for the constructor (defconstr (mypair myfst *)) would be: “mypair.0” and “mypair.1”. After those names have been created, the selectornames as given in the file are read. In the example these would be “myfst” and “*”. All “*”s are then replaced with the constructor names as created, and the rest is left as they were given. The resulting selector names for our example are “myfst” and “mypair.1”. Once these names are constructed, some dynamic rules are created that link those names to the indices of the arguments from a constructor which they refer to.

```
ConstrSelRules :
  Loadt(filename) ->
  Loadt(filename)
    where <ParseSimilix>filename => contents
        ; < ?OCDs(<id>)
          ; filter(?Defconstr(<id>))
          ; concat>contents => temp
        ; map(MkConstrSelNames) => constrselnames
        ; <map(Snd)>temp => temp2
        ; < zip(zip(Ss2OrC))
          ; MkConstrSelRules>(temp2,constrselnames)

MkConstrSelRules :
  a -> a
  where < zip(id)
        ; map(zip(id))
        ; concat
        ; map(MkConstrSelRule)>
            (a,<map(length;dec;upto;map(inc))>a)

MkConstrSelRule :
  (a,b) -> (a,b)
  where rules(LookupS : S(a) -> b)

Ss2OrC :
  ("*",b) -> b

Ss2OrC :
  (Ss2(a),b) -> a

MkConstrSelNames :
  (a,b) -> constrselnames
    where < length
          ; dec
          ; upto
          ; map(int-to-string)>b => nums
        ; <zip(ConcatNames)>( <map( \ x -> a \ )>nums
                            , nums) => constrselnames

ConcatNames :
  (a,b) -> <concat-strings>[a,".",b]
```

6.2 Strategy used for Binding Time Analysis

During binding time analysis of a procedure, the first thing to do, after the preparations, is to look for the procedure to be analysed. The procedure to be analysed and the values for which it is to be analysed have to be given in the procedure named "bta", i.e., there has to be a function somewhere in the program which looks like this:

```
(define (bta)
  (function-to-be-analysed arg1 arg2 ...))
```

The arguments to this function can either be values (for example: 1, 'a', (makepair 1 2), '("a" "b" "c")), variables, or partial values (for example: (makepair 1 a)).

Binding time analysis is started by looking for the function named "bta". Once this function is found, its body is inspected, which should be an application of a function to a list of arguments. A dynamic rule is then created which adds this list of arguments to the function definition. This function definition now is the only function definition which has a tuple of arguments. The first element of this tuple is the list of names of the arguments, the second element is the list of values for which binding time analysis has to be done. This function is looked up and analysed, as described in Section 6.3.

During this analysis function calls can be met, with arguments which have annotations. The definitions of those functions are then looked up, arguments are added, and they are analysed as well. Because it is possible that functions are recursive, and because the annotations of the result of a function call are needed to determine the annotations of the function it is called from, a recursion-check is done before annotation is started. This recursion check maintains a list of functions with annotated arguments which are currently being worked on. If a function passed to the recursion check already is in the list with the same set of annotations for the arguments, the recursion check reports that the function is recursive, and the function call is annotated with a special "Rec"-annotation. Otherwise, the function is just analysed in the normal way.

When a function definition is analysed in a normal way, the whole body is annotated. A special function ProcAnno is created, which gives the annotation of the result of a function call. This function can be used when the same function with the same annotations is come across later. For example in the program:

```
(define (special-addition a b c)
        (+ (plus a b)
           (plus b c)))

(define (plus a b) (+ a b))

(define (bta)
        (special-addition 1 2 3))
```

the function plus is called twice, both times with both arguments being static. After an analysis is done for the function plus once, it is known that the result is static, and it is not necessary to do this analysis again for the second call. The function ProcAnno is used the second time.

When a function is analysed in a normal way, the result of this analysis is put in a list, which consists of the not-analysed version of this function followed by all analyses come across so far. A rule is created which returns this list given the not-analysed version of the function. This function is used in the next step of binding time analysis, which replaces all not-analysed functions in the program by the list returned by AnnotateDefinition, or, if AnnotateDefinition does not exist for a function definition, puts this function definition in a list. All those lists are then concatenated, resulting in the original program, with added annotated definitions.

The final step of binding time analysis sees if there are any annotations "Rec" in the program. When there is a annotation "Rec" in the program, this means that no resulting annotation for the function call could be computed, as this function was already being analysed. Because binding time analysis is finished, these resulting annotations are available now, and can be used to remove all remaining “Rec”-annotations.

```
strategies
  startbta =
    prepare
  ; where( ?Program(<map(OfaOvaRules)>,_,_))
  ; where( <table-create>"defs"
         ; <table-create>"pending"
         ; <table-put>("pending", "pending", [])
         )
  ; where( ?Program( _
                   , < fetch(LookupBTAfunction)
                     ; filter(ToBTA)
                     ; ?[<id>]>
                   ,  _)
         ; bta
         )
  ; Program( id
           , map(AnnotateDefinition <+ ![<id>]); concat
           , id) => p
  ; (
      ( <eq> (<collect-annos(Rec)>p,[]); !p )
      <+ bottomup(try(AnnotateRec))
    )


  bta =
  ; recursion-check
  ; try( ?Define( name@Name(realname,"")
                , (namedargs,valueargs)
                , body)
       ; btastrategy
       ; ?Define( _, _, body2 )
       ; <?Body([],[<GetAnnotation>])>body2 => anno
       ; <map(Annotate;RmVals)>valueargs => annos
       ; <FinishBTA>Define( name
                          , (namedargs,valueargs)
                          , body2) => btaddefine
       ; where( !(btaddefine,body)
              ; updatetables)
       ; rules(
           ProcAnno :
             App(P(realname),annos) -> anno

           AnnotateDefinition :
     Define(name, argsa, body) ->
             <table-get>("defs", (name,argsa) )
              )
       )

  recursion-check =
    ?Define( name@Name(realname,"")
            , (namedargs,valueargs)
            , body ) => define
  ; !App( P(realname)
        , <zip(MoveAnnos)>( namedargs
                          , <map(Annotate)>valueargs)) => app
  ;  ( alreadypending; !define)
    <+ !BTAd(Define("name","args",Body([],["expr"{Rec}])))

  updatetables =
    ?(btaddefine@BTAd(Define(name@Name(realname,_),namedargs,_),body))
  ; <map(RmAnnos)>namedargs => args
  ; ( <table-get> ( "defs", (Name(realname,""),args))
   <+ ![Define(Name(realname,""),args,body)]) => list
  ;  <table-put> ( "defs", (Name(realname,""),args)
                       , <conc>(list, [btaddefine]) )

  alreadypending =
    ?app
  ; <table-get>("pending","pending") => pendinglist
  ; <collect(\x -> x where <eq>(x,app) \ )>
        pendinglist => alreadypending
  ; <eq>(alreadypending,[])
  ; <table-put>("pending","pending",[app | pendinglist])
```

6.3 Analysis of a single function

To analyse a single function, first all known annotations are distributed through the program, and then use those annotations to compute the annotations that are not known yet. The distribution of annotations through the program is explained in Section 6.3.1, the computation of annotations is explained in Section 6.3.2.

6.3.1 Distribution of annotations

In this section an explanation of the distribution of annotations of bound variables is explained. To do this distribution the following strategy is used:

```
  btastrategy =
    rec distributeannos
      ({| AnnotateLetBoundVars
        , AnnotateVars
        : try( Let([(id,distributeannos)],id)
             + App(Lambda(id,id),distributeannos))
             ; try( LetVarRuleGenerator
                  + VarRuleGenerator
                  + AnnotateLetBoundVars
                  + AnnotateVars
                  + LambdaVarRuleGenerator
                  + MkConstantsStatic
                  )
        ; ( Let([(distributeannos,id)],distributeannos)
          <+ App(Lambda(id,distributeannos),id)
          <+ Define(id,id,distributeannos)
          <+ all(distributeannos))
        ; bottomupannotate
       |})
```

Annotation of constants

All constants are made static with the rule MkConstantsStatic.

Annotation of variables passed as arguments

The values or annotations of arguments of a to be analysed procedure are always known and their annotations can be distributed through the program. That is done using dynamic rules. A rule VarRuleGenerator is available, which annotates the values passed to the procedure, and creates dynamic rules which annotate a variable with its annotation.

```
VarRuleGenerator =
  ?Define(name,(namedargs,valueargs),body);
      where( <map(Annotate)>valueargs
                 => annotatedvalueargs
           ; <zip(MkVarAnnotationRules)>( namedargs
                                        , annotatedvalueargs))

      MkVarAnnotationRules =
        ?(var,val{anno});
            where (rules(AnnotateVars :
                           V(var) ->
                           V(var){anno}))
```

Annotation of let-bound variables

To determine the annotations of let-bound variables the expressions to which the variables are bound have to be annotated first (see btastrategy: Let([(id,distributeannos)],id)). These annotations are then distributed to the let-bound variables and a dynamic rule is created which annotates a variable with its annotation.

```
   LetVarRuleGenerator :
    Let([(var,val{anno})],body) ->
    Let([(var{anno},val{anno})],body)
      where <is-string>var
          ; rules(AnnotateLetBoundVars :
                    V(var) ->
                    V(var){anno})
```

Annotation of lambda-bound variables

The values to which the variables in a lambda-expression refer are annotated first. (see btastrategy: App(Lambda(id,id),distributeannos)) The annotations of those values are then transferred into the lambda-expression, and the same dynamic rules used for annotating define-bound variables are used to distribute the annotations through the program.

```
  LambdaVarRuleGenerator :
    App(Lambda(vars,body),vals) ->
    App(Lambda(annvars,body),vals)
        where <zip(MoveAnnos)>(vars,vals) => annvars
            ; <zip(MkVarAnnotationRules)>(vars,vals)
```

6.3.2 Computation of annotations

Almost all bottom elements in the program are now annotated, that is, all bound variables and all constants. The rest of the annotations can now be computed from those bottom elements, using the following strategy. First it is checked if the expression currently to be annotated is an application of a lambda-expression, constructor-test, selector or a constructor. These are all applications of special expressions to arguments. If the expression currently to be annotated is such an application, the expression is annotated using the rules existing for those application. When those rules fail, all other rules, including the one that annotates all other applications, can be tried.

```
bottomupannotate =
  try( ( AnnotateLambda
       + AnnotateConsQ
       + AnnotateSel
       + AnnotateConstr)
     <+ AnnotateApp
      + ( AnnotateIfRec
       <+ AnnotateIf)
      + AnnotateLet
      + AnnotateLis
      + AnnotateSeq
      + AnnotateVar)
```

Annotation of variables

All variables left in the program that do not have annotations yet, are free variables. These variables are annotated as dynamic.

```
AnnotateVar :
  V(x){} ->
  V(x){Dy}
```

Annotation of sequences

The result of doing one thing after another is, of course, the result of the second thing. Therefore the annotation of a sequence is also the annotation of the second element of the sequence.

```
AnnotateSeq :
  Seq([a,b{anno}]) ->
  Seq([a,b{anno}]){anno}
```

Annotation of lists

A list is annotated with a special List-annotation, LisAnno. All elements in the list are annotated, and those annotations are put in a list, which is used as the annotation of the list. The special LisAnno is used to distinguish between annotations of lists, and annotations of constructors. It is also useful when lists of lists are annotated.

```
AnnotateLis :
  Quote(Lis1(x){anno}) ->
  Quote(Lis1(x)){anno}
AnnotateLis :
  Lis1(x) ->
  Lis1(x){LisAnno(anno)}
    where <map(RmVals)>x => anno

AnnotateLis :
  Quote(Lis2(x,y){anno}) ->
  Quote(Lis2(x,y)){anno}

AnnotateLis :
  Lis2(x,y) ->
  Lis2(x,y){LisAnno(anno)}
    where <map(RmVals)>[x,y] => anno
```

Annotation of constructors

A constructor is annotated with a list of annotations. The first element of the list indicates whether the constructor is known, the rest of the list consists of the annotations of the arguments of the constructor.

```
  AnnotateConstr :
    App(C(x),args) ->
    App(C(x),args){annargs}
      where [St | <map(RmVals)>args] => annargs
```

Annotation of constructor-tests

As explained in the previous section, the first element of the list of annotations of a constructor indicates whether the constructor is known. This means that if the annotation of a constructor-test is needed, all that is to be done, is to look at the head of the list of annotations of the argument.

```
  AnnotateConsQ :
    App(ConsQ(x),[arg]) ->
    App(ConsQ(x),[arg]){anno}
      where <GetAnnotation;Hd>arg => anno
```

Annotation of selectors

A selector selects a certain element from a constructor. Because rules to decide which element which selector selects are already created in the preparation phase, the index of the element which this selector selects can be looked up. As the first element in the list of annotations indicates whether the constructor is known, the first element in the list has to be skipped, and the index which has been looked up has to be increased. It is now a question of selecting the right annotation from the list of annotations.

```
  AnnotateSel :
    App(S(x),[arg{arganno}]) ->
    App(S(x),[arg]){anno}
      where <index>(<LookupS;inc>S(x),arganno) => anno
```

Annotation of let-expressions

The result of a let-expression is the result of its body, and therefore the annotation of a let-expression is the annotation of its body as well.

```
  AnnotateLet :
    Let([b],Body([],[x{anno}])) ->
    Let([b],Body([],[x{anno}])){anno}
```

Annotation of lambda-expressions

The result of a lambda-expression is the result of its body, and therefore the annotation of a lambda-expression is the annotation of its body as well.

```
  AnnotateLambda :
    App(Lambda(a,Body([],[b{anno}])),args) ->
    App(Lambda(a,Body([],[b{anno}])),args){anno}
```

Annotation of if-expressions

If one branch of an if-expression has a recursive call, and the other has not, then the annotation of the none-recursive branch is chosen, as recursion can only end in a choice, and an if-expression is the only choice available in core similix. The condition has to be annotated static, else the choice can not be made, and partial evaluation of the recursive function call would go on forever, as both branches are partially evaluated.

If the condition is annotated static, then it is known that at evaluation time it is known which branch is chosen. As it is not known which branch is chosen during binding time analysis, something can only be said about the result of an if-expression if the condition is annotated static and both branches are annotated the same.

If the condition is not annotated static, or the annotations of the branches differ, the result of the if-expression is unknown and thus dynamic.

```
AnnotateIf :
    If(a{St},b{anno},c{anno}) ->
    If(a{St},b{anno},c{anno}){anno}

  AnnotateIf :
    If(a{condanno},b{banno},c{canno}) ->
    If(a{condanno},b{banno},c{canno}){Dy}
      where (not(<eq>(condanno,St)) + not(<eq>(banno,canno)))

  AnnotateIfRec :
    If(a{St},b{annob},c{annoc}) ->
    If(a{St},b{annob},c{annoc}){anno}
      where <filter(not(Rec))>[annob,annoc] => [anno]
```

Annotation of an application

An application is annotated with the annotation of the result of that application. If the application is an application of a procedure, the name of that procedure has to be changed to the name of the annotated variant of that procedure.

If an application has to be annotated, first it is checked if the rule ProcAnno, which is created when a procedure is applied to some arguments exists. If the application that is to be annotated now is an application of the same procedure with the same annotations for its arguments, the annotation does not have to be computed again, and the rule ProcAnno can be used.

If an application of a procedure is to be annotated, which has not been annotated before, an annotated version of that procedure is created and the annotation of the body of that annotated version is used as the annotation of the procedure application (GetSpecificAnnos).

Some applications need special treatment. For example, the maximum element of a list can only be determined if all elements in that list are known.

If an application is neither an application of a procedure, nor an application which needs special treatment, the general rules for annotating applications are used. Those general rules say that the result of an application is only static if all of its arguments are static.

```
AnnotateApp :
  app@App(x,args) ->
  app{<GetAnnos>app}
    where not(<?P(_)>x)

AnnotateApp :
  app@App(P(x),args) ->
  App(P(<BtaName>app),args){<GetAnnos>app}

GetAnnos = RecAnno
        <+ ProcAnno'
        <+ GetSpecificAnnos
        <+ GetGeneralAnnos

ProcAnno' :
  App(P(func),args) -> anno
    where <map(Annotate;RmVals)>args => args'
        ; <ProcAnno>App(P(func),args') => anno

RecAnno :
  App(func,args) -> Rec
    where <filter(?_{Rec})>args => [_|_]


GetGeneralAnnos :
  App(func,args) ->
  St
    where <eq>(<filter(?_{St})>args,args)

GetGeneralAnnos :
  App(func,args) ->
  Dy
    where not(<eq>(<filter(?_{St})>args,args))
        ; <eq>(<filter(?_{_})>args,args)

GetSpecificAnnos :
  App(P(x),args) ->
  annos
    where <LookupFunction>( Name(x,"")
                          , <length>args) => (args2,body)
        ; <bta>Define( Name(x,"")
                     , (args2,<map(Annotate)>args)
                     , body) => (_,_,res)
        ; < reverse
          ; Hd
          ; ?BTAd(Define( _
                        , _
                        , Body(_,[<GetAnnotation>])))>res => annos

GetSpecificAnnos :
  App(Ova("max"),x) -> St
    where <GetAnnotation;collect(Dy)>x => []
```

Chapter 7

Binding Time Improvements

The goal of this study is not to do as many binding time improvements as possible, but as an illustration how easy binding time improvements can be done, some very simple ones will be shown here.

As it would be a waste to be unable to calculate the result of an addition when only a few of the arguments are dynamic, whilst most are static, the arguments are split into two lists. The first list contains all static arguments, and the second all dynamic ones. An static addition can now be made with all static arguments, and this addition can be put as an argument to a dynamic addition of all dynamic variables.

For a subtraction, the approach is the same, but all static arguments have to be added, and this addition has to be subtracted from the first argument together with all dynamic arguments.

```
a - 2 - 3 - 5 - b = a - (2 + 3 + 5) - b
```

When the first argument is static, an even further reduction can be achieved:

```
12 - 2 - b - 4 - 8 = (12 - (2 + 4 + 8)) - b
```

```
strategies
   startbti = topdown(try(BTICalc))

   Similix-BTI = iowrap(startbti)


rules

  BTICalc :
   App(Ova(x),args){Dy} ->
   App(Ova(x),<conc>( [dy|dys]
                    , [App(Ova(x),[st|sts]){St}])){Dy}
     where (<eq>(x,"*") + <eq>(x,"+"))
         ; <collect(?_{Dy})>args => [dy|dys]
         ; <collect(?_{St})>args => [st|sts]

  BTICalc :
   App(Ova(x),[arg|args]){Dy} ->
   App(Ova(x),<conc>( [arg|dys]
                    , [App(Ova(<inv>x),[st|sts]){St}])){Dy}
     where (<eq>(x,"-") + <eq>(x,"/"))
```

```
       ; <get-annotations>arg => [Dy]
       ; <collect(?_{Dy})>args => dys
       ; <collect(?_{St})>args => [st|sts]

BTICalc :
 App(Ova(x),[arg|args]){Dy} ->
 App(Ova(x),<conc>( [App(Ova(x),[ arg{St}
                                 , App( Ova(<inv>x)
                                      , [st|sts]){St}]){St}]
                  , dys))
   where (<eq>(x,"-") + <eq>(x,"/"))
       ; <get-annotations>arg => [St]
       ; <collect(?_{Dy})>args => dys
       ; <collect(?_{St})>args => [st|sts]

inv : "-" -> "+"
inv : "/" -> "*"
```

Chapter 8

Partial Evaluation

Almost the same structure as was used for binding time analysis is used for partial evaluation. The only differences are that no recursivity checks have to be done, and that an other solution has to be found for partial static values, as will be seen later in this chapter.

8.1 Structure used for partial evaluation

Before anything is done to partially evaluate functions, a topdown traversal is done, which gives all constants its real value. A number is, for example, stored as Num("1"). During partial evaluation some computations have to be done on such values, and to prevent unnecessary complicated rules, which rewrite the string "1" to the number 1 and rewrite the result of the computation to a string again, one top-down traversal is done that makes numbers from all strings in a Num-constructor, makes characters from all strings in a Char-constructor and maker strings from all strings in a Str-constructor (a string "abc" was represented as ""abc"").

After all constants are rewritten to their values, the same preparations that were done for binding time analysis are done again. Remember: functions can be looked up now, and rules are made that connect a selector to the index it selects.

The rules for Ofas and Ovas are defined (see Section 8.1.1), the function named "bta" is looked up, arguments are added, and the function is passed to the "pe" strategy.

In the end, partially evaluated versions are added to all functions for which partially evaluated versions are made, and all values are rewritten to their string-representations again.

```
strategies

  Similix-PE =
    iowrap(startpe)

  startpe =
    topdown(try ( NumToNum
                + CharToChar
                + StrToStr))
    ; prepare
    ; where( <table-create>"defs"
           ; <table-create>"pending")
    ; where( ?Program(<map(OfaOvaRules)>,_,_) );
    ; where( ?Program( _
                    , < fetch(LookupBTAfunction)
                      ; filter(ToPE)
                      ; ?[BTAd(<id>)]>
                    , _)
          ; pe)
    ; Program( id
             , map(FillDefinition <+ ![<id>]); concat
             , id);
    ;topdown(try ( NumFromNum
                 + CharFromChar
                 + StrFromStr))
```

8.1.1 Rules for user defined operators

Some rules are created to handle the user defined operators. All Loadt-ed files are parsed, and their contents are filtered for the different possibilities to define operators.

For all operators a rule is created that rewrites the application of the user-defined operator to an application of the definition given by the user for that operator. Those rules are only created when the operator is defined to be reducible (i.e., defined with a defprim, defprim-transparent or defprim-tin).

The KAOVs (Key, Arity, Operator-name, scheme-Variable) and KOVs (Key, Operator-name, scheme-Variable) are filtered out of the contents of the file, and rules are created which rewrite a static application of this operator to a static application of the scheme-variable.

The KOVE1s (Key, Operator-name, Variables, scheme-Expression) are filtered out of the contents of the file, and rules are created which rewrite a static application of this operator to a static lambda-expression having the variables as parameters, and the arguments to the original application as arguments. The body of the lambda expression is the desugared scheme-expression. Desugaring has to be done because in the sugared version all applications mentioned in the scheme expressions are applications of variables to variables, and not applications of, for example, Ofas to variables. The choice is made to do this desugaring only when an application of this operator is come across, even though this means that desugaring has to be done every time this operator is encountered. The alternative would be that all operators mentioned in a loadt-ed file defined through a KOVE1 are desugared once, and in general less occurrences of a KOVE1-defined operator are come across in a program than there are KOVE1-defined operators.

```
  OfaOvaRules =
    ?Loadt(filename);
      where (< ParseSimilix
             ; ?OCDs(<id>)>filename => contents
          ;  < filter( KOVE1(id,id,id,id) )
             ; map(try(MkKOVE1Rules))>contents
          ;  < filter( KAOV(id,id,id,id) + KOV(id,id,id))
             ; map(try(MkKOVRules))>contents)


  desugar = KOVE1ToDefine;try(Desugar)

  KOVE1ToDefine :
    KOVE1(key,name,vars,E(body)) ->
    [Define(name,vars,Body([],[body]))]
```

```
Desugar : [def|defs] -> contents
  where new-file => newfile1
      ; new-file => newfile2
      ; <WriteToTextFile>( newfile1
                         , Program([],[def|defs],[]))
      ; <call>("./Similix-Desugar",[ "-i", newfile1
                                   , "-o", newfile2])
      ; <ReadFromFile> newfile2 => contents
      ; <call>("rm",["-f",newfile1,newfile2])

MkKOVE1Rules =
  ?KOVE1(key,name,vars,body);
  where( <Reducible>key;
         rules( UserDefinedPrim :
                  App(Ofa(name),args){St} ->
                  <MkStatic>App(Lambda(vars,body2),args)
                    where <desugar>KOVE1(key,name,vars,body)
                         => Program([],[Define(_,_,body2)],[])))

MkKOVRules =
  ?KAOV(key,ari,name,Ova(primi));
  where (<Reducible>key;
         rules( UserDefinedPrim :
                  App(Ofa(name),args){St} ->
                  App(Ofa(primi),args){St}))

MkKOVRules =
  ?KOV(key,name,Ova(ova));
  where (<Reducible>key;
         rules( UserDefinedPrim :
                  App(Ova(name),args){St} ->
                  App(Ova(ova),args){St}))

MkStatic :
  App(Lambda( vars
            , Body([],body))
            , args) ->
  App(Lambda( <map(MkStatic)>vars
            , Body([],<bottomup(try(MkStatic2))>body))
            , args){St}

MkStatic :
  a -> a{St} where <is-string>a

MkStatic2 :
  a -> a{St}
    where < ?V(_)
          + ?App(_,_)
          + ?Let(_,_)
          + ?Lambda(_,_)
          + ?Seq(_)>a

Reducible =
  ?key
  ; ( <eq>(key,"defprim")
    + <eq>(key,"defprim-transparent")
    + <eq>(key,"defprim-tin"))
```

8.2 Partial evaluation of a single function

Before the partial evaluation is started for a single version of a function some administration has to be performed for this function. First its partial evaluation name, which holds all information about the input of the function, is added to the table "pending". Then partial evaluation is really done, and after that the administration is updated. The arguments of the partially evaluated function are standardized, i.e., all constants are left as they are, and all other arguments are renamed to variable "x". These standardized arguments are used in the function FillProc which can be used for looking up the result of partially evaluating a specific function for a specific set of arguments. This prevents that a function is partially evaluated for the same set of arguments twice. The result of partially evaluating the function is stored in the table "defs", which is used in the rule FillDefinition, which in turn is used when adding partially evaluated versions of a function to the originals at the end of partial evaluation (see Section 8.1).

```
pe =
  ?Define(name,(namedargs,valueargs),body) => define
; where(updatependingtable)
; pestrategy
; ?Define( name
         , (_, valueargs)
         , body2@Body(_,[innerbody2])) => define2
; <map(MkStandard)>valueargs => args'
; where(!(define2,body);updatedefstable)
; !innerbody2
; rules( FillProc :
           App(P(name), args') -> innerbody2

         FillDefinition :
           BTAd(Define(name,namedargs,body)) ->
           <table-get>("defs",(name,namedargs))
        )

updatependingtable =
  ?Define(name,(namedargs,valueargs),body)
; <PeName>App( P(name)
             , <map(topdown(try( NumFromNum
                               + CharFromChar
                               + StrFromStr)))>valueargs)
        => pename
; <table-put>("pending",pename,pename)


updatedefstable =
  ?(Define(name,(namedargs,valueargs),body2),body)
;  ( <table-get>("defs",(name,namedargs))
  <+ ![BTAd(Define(name,namedargs,body))]) => list
```

```
; <table-put>( "defs"
             , (name,namedargs)
             , <concat>[ [<Hd>list]
                         , [<FinishPE>Define(name
                              , (namedargs,valueargs)
                              , body2)]
                         , <Tl>list])
```

Now all administration is done, real partial evaluation can begin. In anology with the annotations during binding time analysis, the values of the arguments are distributed through the program. Before distribution of variables is started in the normal way, first is checked if the expression through which the variables have to be distributed matches an application of a user-defined operator which can be reduced through one of the rules created earlier. If this is the case this reduction is done.

After this it is tried to determine the value to which a let-bound variable is bound, because the value to which a let-bound variable is bound has to be known before the value of the let-bound variable can be distributed through the program. The values to which lambda-bound variables are bound are attempted to be determined for the same reason. Also, it is tried to determine the value of the conditional expression in an if-expression, as this value is needed to make a choice about which branch to choose later.

```
pestrategy =
  rec distributevals
  ( {| FillVars
     , AddInfo
     , AddInfoRec
     , FillLetBoundVars
     , AddInfoLetBoundVars
       : try(UserDefinedPrim)
       ; try( Let([(id,distributevals)],id)
            + App(Lambda(id,id),distributevals)
            + If(distributevals,id,id))
       ; try( VarRuleGenerator
            + AddInfo
            + AddInfoRec
            + FillVars
            + LetVarRuleGenerator
            + AddInfoLetBoundVars
            + LambdaVarRuleGenerator
            + FillLetBoundVars
            )
       ; ( Let([(distributevals,id)],distributevals)
        <+ App(Lambda(id,distributevals),id)
        <+ Define(id,id,distributevals)
        <+ IfTrue(distributevals)
        <+ IfFalse(distributevals)
        <+ Info(id,id)
        <+ all(distributevals));
        <+ bottomupevaluate
    |})
```

After the values are distributed through the expression currently in treatment, it is needed to indicate to which children of this expression the values need to be distributed. As the values of variables are already distributed through the value-part of let-expressions and lambda-expression, it makes no sense to do that again. Also, it is not necessary to distribute any values to the name or arguments of a function definition, nor is it necessary to distribute any values into an Info-expression (see Section 8.2.1). However, when an if-statement is come across, it may be possible to discard one of the branches depending on the value of the conditional expression.

```
  IfTrue(s) :
    If(Bool("#t"),thenpart,elsepart){annot} ->
    If(Bool("#t"),<s>thenpart,0){annot}

  IfFalse(s) :
    If(Bool("#f"),thenpart,elsepart){annof} ->
    If(Bool("#f"),0,<s>elsepart){annof}
```

If the conditional expression of the if-expression is evaluated to #t, there is no need to distribute any values through the else-branch, and values need to be distributed values through the then-branch only. The else-branch can be deleted. Because a if-expression needs three branches, the else-branch is simply replaced by zero. If the conditional expression of the if-expression equals #f, the then-branch can be discarded, and values distributed through the else-branch. Id the conditional expression is neither #t nor #f, nothing can be said about which branch to choose, so values need to be distributed through both branches.

8.2.1 Distribution of values of arguments through a function

When the values of arguments of a function have to be distributed through the program, there are two possibilities. The first possibility is that a value is totally static. In that case, the variable referring to that value can be replaced by that value. This means that for all arguments whose annotations contain no dynamics, a rule is created which replaces the variable mentioned in the argument with the value. The second possibility is that a value is partially static. For example, a list is passed to the function as an argument : '(1 2 a). It can be seen that the first two elements of the list are static, and the rest of the list is dynamic. In this case, the variable referring to that list can not just be replaced by the list, as the list is no value that can be mentioned in a resulting program. For example, the function:

```
(define (function a) a)
```

with input '(1 2 a) returns '(1 2 a), which is no valid output.
But, when the function:

```
(define (function a) (car a))
```

is to be specialized for the input '(1 2 a), the output of this function is 1. So, a mechanism is required that leaves the variable as it is, but gives access to the partial static value of that variable if needed. To achieve this, all that has to be done is that if a value is annotated with statics, but not only with statics, the variable referring to that value has to be replaced by Info(variable,value). If the value is needed, it can be found in the second element of the Info, but when partial evaluation is finished, only the first element of the Info has to be returned.

```
  VarRuleGenerator :
    Define(name,(namedargs,valueargs),body) ->
    Define(name,(namedargs,valueargs),body)
      where < zip(AllStatic)
            ; filter(not(is-string))
```

```
          ; map(MkFillVarRules)>(namedargs,valueargs)
        ; < zip(ExistsStatic)
          ; filter(not(is-string))
          ; map(MkAddInfoRules)>(namedargs,valueargs)

MkFillVarRules :
  (var,val) ->
  (var,val)
    where rules(FillVars : var -> val)

MkAddInfoRules :
  (var,Info(var2,val)) ->
  (var,Info(var2,val))
    where rules(AddInfoRec : var -> Info(var,val))

MkAddInfoRules :
  (var,val) ->
  (var,val)
    where not(<?Info(_,_)>val)
        ; rules(AddInfo : var -> Info(var,val))

AllStatic :
  (x{annos},b) ->
  (V(x{}){annos},b)
    where <collect(Dy)>annos => []

AllStatic :
  (x{annos},b) -> x
     where not(<eq>(<collect(Dy)>annos,[]))

ExistsStatic :
  (x{annos},b) ->
  (V(x{}){annos},b)
    where not(<eq>(<collect(St)>annos,[]))
        ; not(<eq>(<collect(Dy)>annos,[]))

ExistsStatic :
  (x{annos},b) -> x
    where ( <eq>(<collect(St)>annos,[])
          + <eq>(<collect(Dy)>annos,[]))
```

8.2.2 Distribution of let-bound variables

When a let-expression is encountered and the value to which the variable is bound by this let-expression is only annotated with statics, the variable can be replaced by the value. All occurences of this variables, as well in the let-expression itself as in the rest of the program, are replaced by this value using a dynamic rule. When the value to which the variable is bound by this let-expression is partially static, the variable has to be replaced by an Info(variable,value)-pair.

```
LetVarRuleGenerator :
  Let([(var{anno},val)],body) ->
  Let([(val,val)],body)
    where <AllStatic>(var{anno},val) => (var',val)
```

```
         ; rules(FillLetBoundVars :   var' -> val)

  LetVarRuleGenerator :
    Let([(var{anno},val)],body) ->
    Let([(Info(var{},value),val)],body)
      where <try(?Info(_,<id>))>val => value
          ; <ExistsStatic>(var{anno},value) => (var',value)
          ; rules(AddInfoLetBoundVars : var' -> Info(var'{},value))
```

8.2.3 Distribution of lambda-bound variables

Lambda-bound variables are distributed through the program in exactly the same way as the arguments of the function were.

```
  LambdaVarRuleGenerator :
    App(Lambda(vars,body),vals) ->
    App(Lambda(vars,body),vals)
      where < zip(ExistsStatic)
            ; filter(not(is-string))
            ; map(MkAddInfoRules)>(vars,vals)
          ; < zip(AllStatic)
            ; filter(not(is-string))
            ; map(MkFillVarRules)>(vars,vals)
```

8.2.4 Bottomup computation of values

After all values have been distributed through the program, those values can be used to do some computations. If a let-expression having the same value in its variable part as in its valuepart is come across, it can be removed, as it does not bind any variables anymore. All parameters and arguments of a lambda-expression that are no variables anymore, but are constants instead, can be removed as well. If no parameters and arguments are left after this removal, the entire lambda-expression can be removed.

```
strategies
  bottomupevaluate =
    try( ( RemoveIdLets
         + RemoveLambdas)
      <+ ( FillApp
         + FillIf)
      <+ RmAnnos)


rules
  RemoveIdLets :
    Let([(a,a)],Body([],[body])) -> body

  RemoveLambdas = RemoveLambdas1;try(RemoveLambdas2)

  RemoveLambdas1 :
    App(Lambda(args,body),vals) ->
    App(Lambda(args',body),vals')
      where <filter(NoDys)>args => args'
          ; <filter(NoDys)>vals => vals'

  RemoveLambdas2 :
    App(Lambda([],Body([],[body])),[]) ->
    body

  NoDys :
    x{anno} -> x{}
      where <collect(Dy)>anno => [_|_]
```

Partial evaluation of if-expression

An if-expression with a #t as conditional expression can be replaced by its then-part. An if-expression with a #f as conditional expression can be replaced by its else-part.

```
  FillIf :
    If(Bool("#t"),thenpart,_){annos} ->
    thenpart

  FillIf :
    If(Bool("#f"),_,elsepart){annos} ->
    elsepart
```

Partial evaluation of applications

When a static application is come across, which has no sequences in its arguments, the application can be replaced by its result. When a static application is come across, which has sequences in its arguments, the application can be replaced by the result of the application to the last element of the sequences, preceded by the first elements of the sequence. For example:

```
(define (plus)
  (+ (begin (display ‘‘1’’)
            1)
     (begin (display ‘‘2’’)
            2)))
```

is reduced to:

```
(define (plus)
   (begin (display ‘‘1’’)
          (begin (display 2)
                 3)))
```

Because the display-expression can not just be thrown away, they have to be lifted out of the application.

When a dynamic function-call is come across, first its partial-evaluated name is created, which holds all information about the input of the function. When this name is not in the table “pending” yet, that means that no partial evaluated version for this input is (being) created yet, the partially evaluated version has to be created. This partially evaluated version is only created if it is not a recursive call to a function seen before, without any hope that this recursion will terminate for the given input (see Section 8.3.The function-call can be replaced by the call to the partial evaluated version of this function.

When a function-call is dynamic, this only means that the result of this function is dynamic, and not that the function can not be partial evaluated. For example, when this function is called with a being 1 and b being dynamic:

```
(define (function a b) (if (= a 0) (- b 1) (function (- a 1) b)))
```

two versions of the function are created:

```
(define (function-1-_ b) (function-0-_ b))

(define (function-0-_ b) (- b 1))
```

Note that the result of the function function-0-_ is dynamic, but still a lot of the code can be discarded.

```
  FillApp :
    App(x,y){annos} ->
    <GetRes>App(x,y){}
      where <collect(Dy)>annos => []
          ; <filter(?Seq(_))>y => []

  FillApp :
    App(x,y) ->
    seq
      where < filter(?Seq(_))>y => [a | b]
          ; < map(try(?Seq([_,<id>])))>y => y2
          ; < map(try(?Seq([<id>,_])))
            ; foldr( <GetRes>App(x,y2)
                   , ( \ (a,b) -> Seq([a,b]) \ ))>y => seq

  FillApp :
    App(P(x),y){annos} ->
    App(P(z),y2){}
      where <collect(Dy)>annos => [_|_]
          ; <PeName>App( P(x)
                       , <map(topdown(try ( NumFromNum
                                          + CharFromChar
                                          + StrFromStr)))>y) => z
          ; <filter(not(Constant))>y => y2
          ; <table-get>("pending",z) => z

  FillApp :
    App(P(x),y){annos} ->
    App(P(z),y2){}
      where <collect(Dy)>annos => [_|_]
          ; <PeName>App( P(x)
                       , <map(topdown(try ( NumFromNum
                                          + CharFromChar
                                          + StrFromStr)))>y) => z
          ; <filter(not(Constant))>y => y2
          ; not(<eq>(<table-get>("pending",z),z))
          ; <LookupFunction>(x,<length>y) => (namedargs,body)
          ; <endingrecursion>([x],body)
          ; <pe>Define(x,(namedargs,y),body)
```

To get the result of an application a set of GetRes-rules exists.

When the application is an application of a constructor-test, the argument to this application has to be a constructor application, else the result will be #f anyway. When the argument is a constructor application, the result will be #t if the constructor is the same as the constructor-test. Else it will be #f.

When a procedure application is come across, first it will be checked if the dynamic rule FillProc exists for this procedure application. If this rule exists partial evaluation has been done for this function application before. It is than not necessary to do partial evaluation again. When the procedure has not been partially evaluated before, it is necessary to do it now. Because at the end of partial evaluation the procedure is put in a list, the result of the function GetProcRes will be the result of putting this procedure in this list. The function has to be taken out of this result, and the result can be put as result to the function application.

For each primitive application, a rule is present, that returns the result of this primitive application. Of course, such a rule is not present if the primitive must not be evaluated at partial evaluation time (think of: (display "Hello World"), (read "file"), etc.).

For some specific functions, whose result is only a part of the argument, the arguments need not necessarily be completely static. For those functions (for example: selectors, the function "car"), it might be possible to get the result to this function from partially known arguments. To achieve this, a function "GetPartRes" exists. This function returns the first possible partial value it comes across. This means it returns the second element in an Info-construction, the first constructor application, the first cons-,the first append-, or the first list-application it comes across. The Info constructors are only used to attach partial values to variables, so the second element represents the values known for this variable. Cons-, append- and list-applications are the only ways to construct partial static lists.

```
GetRes :
  App(ConsQ(x),[App(C(y),args)]) ->
  <FailToBool(<eq>(x,y))>0

GetRes :
  App(ConsQ(x),[arg]) ->
  Bool("#f")
    where not(<?App(C(_),_)>arg)

FailToBool(s) = s; !Bool("#t") <+ !Bool("#f")

GetRes :
  app@App(P(x),args) ->
  <FillProc' <+ GetProcRes>app

FillProc' :
  App(P(x),args) ->
  <FillProc>App(P(x),<map(MkStandard)>args)


GetProcRes :
  App(P(x),args) ->
  pebody
    where <LookupFunction>(x,<length>args) => (namedargs,body)
        ; <pe>Define(x,(namedargs,args),body) => (_,_,res)
        ; <Tl;Hd;?PEd(Define(_,_,Body(_,[<id>])))>res => pebody

GetRes :
  App(Ofa("abs"),[Num(x)]) ->
  Num(<mul>(-1,x))
    where <lt>(x,0)

GetRes :
  App(S(x),[arg]) ->
  <index>(<LookupS>S(x),<try(GetPartRes);?App(C(_),<id>)>arg)

GetRes :
  App(Ofa("car"),[arg]) ->
  <GetRes>App(Ofa("car"),[partres])
    where not(< ?App(Ofa("cons"),[_,_])
              + ?App(Ova("append"),_)
              + ?Quote(Lis1(_))>arg)
        ; <try(GetPartRes);topdown(try(RmAnnos))>arg => partres

GetPartRes :
  App(P(x),args) ->
  <try(GetPartRes)>pebody
    where <GetRes>App(P(name),args) => pebody

GetPartRes :
  Info(a,b) ->
  <try(GetPartRes)>b

GetPartRes :
  Let(bindings,Body([],[body])) ->
  <try(GetPartRes)>body

GetPartRes :
  App(Lambda(bindings,Body([],body)),args) ->
  <try(GetPartRes)>body

GetPartRes :
  App(Ofa("cons"),[x,y]) ->
  App(Ofa("cons"),[x,y])

GetPartRes :
  App(Ova("append"),x) ->
  App(Ova("append"),x)

GetPartRes :
  App(Ova(‘‘list’’),x) ->
  Quote(Lis1(x))

GetPartRes :
  App(C(x),y) ->
  App(C(x),y)
```

8.3 Termination

When a partially evaluated version of a function application whose result is dynamic is to be created, it is possible that this function application is a recursive function call, which will result in a recursive function call again. When these recursive function calls eventually result in a non-recursive case, as they would in a normal program, this recursion will end. As not all input to the function call is known, it is possible that although the program would end its recursion when all input is known, it will not now. The partial evaluator will then create an infinite number of partially evaluated funtions, and therefore never terminate.

In some cases, it is possible to detect this non-termination, and stop partial evaluation. The rules used here will succeed if there is any hope that partial evaluation will terminate. For some programs the partial evaluator will have hope to terminate forever, and the partial evaluation process still will not terminate. Using this rules guarantees that if there is a terminating partial evaluation, it will be returned. It does however not guarantee that partial evaluation will end.

The rule endingrecursion gets a list of procedure names of functions it has already checked. The names are the names as they were created during binding time analysis. The rule also gets the expression it has to check for ending recursion. Endingrecursion1 says that if an expression contains no function calls, recursion will end in that expression.

The rule endingrecursion2 says that if the expression contains an if-expression with a static conditional element, there is hope to end recursion if one of the two branches stops recursion. The conditional element has to be static, because it is necessary to be able to make the choice, because else the possibly never ending recursive function call in the other branch has to be partially evaluated anyway.

The rule endingrecursion3 says that if there are any if-expression in which both branches end recursion, there is a possibility that recursion ends.

Endingrecursion4 says that if there are no if-expression, but there is an application of a procedure which has not been partially evaluated before and may end recursivity, there is a possibility that recursion ends.

```
endingrecursion =
  endingrecursion1 <+
  endingrecursion2 <+
  endingrecursion3 <+
  endingrecursion4

endingrecursion1 =
  ?(names,body)
  ; <collect-all(App(P(id),id))>body => []

endingrecursion2 =
  ?(names,body)
  ; <collect-all(If(( \ x{anno} -> <eq>(anno,St) \ )
              ,( \ x -> <endingrecursion>(names,x) \ )
                   ,id) +
                 If(( \ x{anno} -> <eq>(anno,St) \ )
                   ,id
                   ,( \ x -> <endingrecursion>(names,x) \ )))>body
                                  => [_|_]

endingrecursion3 =
  ?(names,body)
  ; <collect( If(id,id,id))
            ; map( \ If(a,b,c) -> ((names,b),(names,c)) \ )
            ; concat>body => ifs
  ; <fetch((endingrecursion,endingrecursion))>ifs => [_,_]

endingrecursion4 =
  ?(names,body)
  ; <collect(If(id,id,id))>body => []
  ; <collect(App(P(id),id))>body => apps
  ; <filter(App(P(( \ x ->
       <collect(\ y ->
           <eq>(x,y) \ )>names \ )),id))>apps => temp
  ; <zip(EmptyOrNothing <+ ![]);concat>(temp,apps) => info
  ; <map(Fst;( \ x -> <conc>([x],names) \ ))>info => names'
  ; <map(LookupFunction;Snd)>info => bodies
  ; <zip(id)>(names',bodies) => newnamesbodies
  ; <fetch(endingrecursion)>newnamesbodies => [_|_]

EmptyOrNothing :
  (App(P([]),args),App(P(x),args)) ->
  [(x,<length>args)]
```

Chapter 9

Discussion

9.1 Future work

It would be interesting to implement some more binding time improvements. It is possible that the results get a lot better. Some interesting binding time improvements may be:

- If an equality test is done to make a choice in an if-expression, the equality can be exploited in the then-branch. For example:

  ```
  (if (equal? a:s b:d)
      (f b)
      ....)
  ```

  The function application (f b) can not be reduced, because b is dynamic. But, because it is known that a and b are equal in the then-branch of the expression, (f b) can be replaced with (f a). (f a) can be reduced, as a is static, and the call (f a) becomes static as well.

- When the following expression is encountered in the program:

  ```
  (f (if (choice:d) 3 4))
  ```

  the function application cannot be reduced, because the choice is dynamic. However, it is possible to move the function inside the if-expression. The result will then be:

  ```
  (if (choice:d) (f 3) (f 4))
  ```

  The applications are doubled, but they can both be reduced.

Next to binding time analysis, there may be some other analyses that may give interesting results. For example, when a function:

```
(define (checklist a b) (list? (cons a b)))
```

is partially evaluated for dynamic a and b, the result will be dynamic, while it can be seen by the cons-operator that the result will be a list.

In Similix next to the binding time analysis as presented in this document, some other analyses are done. It would be interesting to see the influence those analyses have on the result of partial evaluation.

Chapter 10

Conclusion

This paper presented an approach to an implementation of the partial evaluator Similix using rewrite rules. The results of partial evaluation match the theoretical results. Comparison with results from Similix show that partial evaluation using Stratego comes close to partial evaluation using Similix. The differences can be explained because Similix uses more binding time improvements and does more analyses. Furthermore, the partial evaluator seems to be quite efficient. Even the more complex programs do not take a long time to be partially evaluated (less than 10 seconds on a Pentium III 500 MHz).

It is proved that the use of rewrite rules is a nice way to describe partial evaluation. All four processes (desugaring, binding time analysis, binding time improvements, partial evaluation) can indeed be split up in single rewriting steps. When those steps, which do not do anything complicated when used alone, are combined using strategies a powerful partial evaluator comes foreward.

Desugaring can be done accumulative, i.e., each construction can be rewritten by a rewrite rule to a more simple construction, which in turn can be rewritten to an even more simple form. Hence, rewrite rules are extremely useful when doing desugaring, as each rewriting to a more simple form can be done with rewrite rules, and the subsequent steps can be combined using rewriting strategies.

Binding time analysis can be done using rewrite rules as well, though they need to be combined with tables to keep the administration up to date. The rewrite rules are not used to do real rewriting, as the program on which they are used stays intact, but are used to add extra information (annotations) to the program. The strategies are used to distribute this information through the program, and to computate annotations from annotations lower in the program. The use of strategies is a nice way of describing this distribution and computation.

A binding time improvement is a rewriting step used on an annotated program. The annotations are used to see how the program can be binding time improved. Each binding time improvement can be expressed using a limited set of rewrite rules, possibly combined using rewriting strategies. The beauty of rewrite rules combined with strategies particularly comes forward in this phase of partial evaluation, as new binding time improvements can easily be added by adding some new rewrite rules.

The phase in which partial evaluation is done can be expressed using rewrite rules, though the administration is kept up to date using tables, as it was done in the binding time analysis phase. The distribution of the values through the program can be expressed in a clearly structured way using dynamic rules, and the computation of derived values can be done in a single bottom-up traversal of the program using normal rewrite rules.

Bibliography

[1] A. Bondorf, Similix 5.0 Manual, DIKU, Department of Computer Science, University of Copenhagen, May 17, 1993

[2] Mikhail A. Bulyonkov and Dmitry V. Kocketov. Practical Aspects of Specialization of Algol-like Programs. In Olivier Danvy, Robert Glück and Peter Thiemann, editors,*Partial Evaluation*, volume 1110 of*Lecture Notes in Computer Science*, February 1996

[3] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, Nicolae Volanschi. A Uniform Approach for Compile-Time and Run-Time Specialization. In Olivier Danvy, Robert Glück and Peter Thiemann, editors,*Partial Evaluation*, volume 1110 of*Lecture Notes in Computer Science*, February 1996

[4] E. Dolstra and E. Visser. Building interpreters with rewriting strategies. In M. G. J. van den Brand and R. Lämmel, editors,*Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, volume 65/3 of*Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002. Elsevier Science Publishers. (To appear)

[5] Neil D. Jones, Carsten K. Gomard and Peter Sestoft. Partial Evaluation and Automatic Program Generation, Prentice Hall International Series in Computer Science, 1993.

[6] N.D. Jones. An Introduction to Partial Evaluation. In ACM Computing Surveys, Vol. 28, No. 3, September 1996.

[7] Jesper Jørgensen, Similix: A self-applicable partial evaluator for Scheme, Department of Mathematics and Physics, Royal Veterinary and Agricultural University, Denmark, 1998, http://www.dina.dk/ jesper/PEsummerschool/Similix/similix.html

[8] Torben AE. Mogensen. Evolution of Partial Evaluators, Removing Inherited Limits. In Olivier Danvy, Robert Glück and Peter Thiemann, editors,*Partial Evaluation*, volume 1110 of*Lecture Notes in Computer Science*, February 1996

[9] K. Olmos and E. Visser. Strategies for source-to-source constant propagation. In B. Gramlich and S. Lucas, editors,*Workshop on Reduction Strategies (WRS'02)*, Electronic Notes in Theoretical Computer Science, Copenhagen, Denmark, 2002. Elsevier Science Publishers. (To appear)

[10] J.D. Stone. The Grinnell Scheme Web, 1995, http://www.math.grin.edu/ stone/scheme-web/

[11] E. Visser. Strategic Pattern Matching. In P. Narendran and M. Rusinowitch, editors *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of*Lecture Notes in Computer Science*, pages 30-44 Tremo, Italy, July 1999. Springer-Verlag.

[12] Eelco Visser. Language independent traversals for program transformation. In Johan Jeuring, editor,*Workshop on Generic Programming (WGP'00)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht.

[13] Eelco Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors,*Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of*Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.

[14] E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System description of Stratego 0.5. In A. Middeldorp, editor,*Rewriting Techniques and Applications (RTA '01)*, volume 2051 of*Lecture Notes in Computer Science*, pages 357-361. Springer-Verlag, May 2001.

[15] Eelco Visser. Scoped dynamic rewrite rules. In Mark van den Brand and Rakesh Verma, editors,*Rule Based Programming (RULE'01)*, volume 59/4 of*Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, September 2001.

[16] E. Visser. The slides covering Stratego from a course on Software Generation, Universiteit Utrecht, September 2001 (http://www.stratego-language.org)

[17] E. Visser. The slides from a course on High-Performance Compilers, Universiteit Utrecht, October 2001 (http://www.stratego-language.org)

[18] J. Visser and J. Scheerder, A Quick Introduction to SDF, CWI, April 25, 2000

[19] MIT Scheme, http://www.swiss.ai.mit.edu/projects/scheme/

[20] Revised(5) Report on the Algorithmic Language Scheme, March 1995, http://www.cs.rice.edu/CS/PLT/packages/doc/r5rs/

Appendix A
Similix.def

```
definition

module Similix
  imports Similix-Whitespace Similix-Program

module Similix-Whitespace
exports
  lexical syntax
    [\ \t\n] -> LAYOUT
    ";"[a-zA-Z0-9\ \t\-\>\(\)\?\'\~
        \"\:\,\.\?\=\|\/\*\#\;\+]*"\n" -> LAYOUT
  context-free restrictions
    LAYOUT? -/- [\ \t\n]

module Similix-Program
  imports Similix-Expressions Similix-OCDs
exports
  sorts Program OCDs
  lexical syntax
  context-free syntax
    OCD+                               -> OCDs {cons("OCDs")}
    TLE* D+ TLE*                       -> Program {cons("Program")}
    "(" "load" F ")"                   -> TLE {cons("Load")}
    "(" "loadt" F ")"                  -> TLE {cons("Loadt")}
    "(" "loads" F ")"                  -> TLE {cons("Loads")}
    "(" "define" "(" P V* ")" B ")"    -> D {prefer,cons("Define")}
    D* E+                              -> B {cons("Body")}

module Similix-Constants
exports
  lexical syntax
    "\""[a-zA-Z0-9\-]+"."[a-zA-Z0-9]+"\""  -> F
    "\""[a-zA-Z0-9\-]+"\""                 -> F
    [a-zA-Z0-9\-\>\*\.\<\[\]\:]+           -> V {avoid}
    [a-zA-Z0-9\-\>\<\=\+\/\*\_\:\.]+       -> P {avoid}
    [a-zA-Z][a-zA-Z0-9\-\>\<\.\=]*         -> C {avoid}
    [a-zA-Z0-9\-\>\.]+                     -> S {avoid}
    "\""[a-zA-Z0-9\-\ \~\:\=\+\*\.]+"\""   -> Str
    [0-9]+                                 -> Num
    "-"[0-9]+                              -> Num
    [0-9]+"."[0-9]+                        -> Num
    "-"[0-9]+"."[0-9]+                    -> Num
    "\'"[a-zA-Z0-9\-\ ]"\'"               -> Char {prefer}
    "#t"                                  -> Bool
    "#f"                                  -> Bool
    [a-zA-Z\-\>\=\*\+\?\/\:\_]+           -> Sym

  context-free syntax

    Bool                  -> SE {cons("Bool")}
    Num                   -> SE {cons("Num")}
    Char                  -> SE {cons("Char")}
    Str                   -> SE {cons("Str")}

    SE                    -> Dat
    Sym                   -> Dat
    Lis                   -> Dat
    Vec                   -> Dat

    SE                    -> K
    "(" "quote" Dat ")"   -> K {cons("Quote")}
    "'" Dat               -> K {avoid,cons("Quote")}

    "(" Dat* ")"          -> Lis {prefer,cons("Lis1")}
    "(" Dat+ "." Dat ")"  -> Lis {prefer,cons("Lis2")}

    "'"Dat                -> Lis {cons("Quote")}
    "#(" Dat* ")"         -> Vec {cons("Vec")}

module Similix-Expressions
imports Similix-Constants Similix-ConditionalExpressions
Similix-LetExpressions Similix-BoolOpExpressions
Similix-CaseExpressions Similix-Ova Similix-Ofa

exports
  context-free syntax
    K                                   -> E
    V                                   -> E {cons("V")}
    C"?"                                -> E {cons("ConsQ")}
    P                                   -> E {avoid,cons("P")}
    C                                   -> E {avoid,cons("C")}
    S                                   -> E {avoid,cons("S")}
    "(" "begin" E+  ")"                 -> E {cons("Seq")}
    "(" "lambda" "(" V* ")" B ")"       -> E {cons("Lambda")}
    "(" E E+ ")"                        -> E {cons("App")}

module Similix-ConditionalExpressions
exports
  context-free syntax
    "(" "if" E E E ")"                  -> E {cons("If")}
    "(" "if" E E ")"                    -> E {cons("IfThen")}
    "(" "cond" ( "(" E E* ")" )+ ")"    -> E {cons("Cond")}
    "(" "cond" ( "(" E E* ")" )*
             "(" "else" E+ ")" ")"  -> E {cons("CondElse")}
```

```
module Similix-LetExpressions
exports
  context-free syntax
    "(" "let"
       "(" ( "(" V E ")" )* ")" B ")" -> E {prefer,cons("Let")}
    "(" "let*"
       "(" ( "(" V E ")" )* ")" B ")" -> E {prefer,cons("LetStar")}
    "(" "let" P
       "(" ( "(" V E ")" )* ")" B ")" -> E {prefer,cons("LetP")}
    "(" "letrec"
       "(" ( "(" P "(" "lambda" "(" V* ")" B ")" ")" )* ")"
           B ")"                      -> E {prefer,cons("LetRec")}

module Similix-BoolOpExpressions
exports
  context-free syntax
    "(" "and" E* ")"   -> E {cons("And")}
    "(" "or"  E* ")"   -> E {cons("Or")}

module Similix-CaseExpressions
exports
  lexical syntax
    "()"       -> MPat {cons("EmptyPat")}
    "_"        -> WiC {cons("ElseWiC")}
    "else"     -> WiC {cons("ElseWiC")}
  context-free syntax
    "(" "casematch"
         E ("(" MPat E+ ")")* ")" -> E {prefer,cons("CaseMatch")}
    "(" "caseconstr"
         E ("(" CPat E+ ")")* ")" -> E {prefer,cons("CaseConstr")}

    "(" MPat E* ")"               -> MPatA {cons("MPatB")}
    MPatA                         -> MPat {avoid,cons("MPatA")}
    K                             -> MPat {cons("K2")}
    "(" MPat "." MPat ")"         -> MPat {prefer,cons("MPat2")}
    V                             -> MPat {avoid,cons("V2")}
    WiC                           -> MPat {cons("WiC")}

    "(" C CPat* ")"               -> CPat {cons("CPat")}
    V                             -> CPat {avoid,cons("V3")}
    WiC                           -> CPat {cons("WiC2")}

module Similix-Ofa
exports
  lexical syntax
    [A-Za-z0-9\-\!\>\*\?\_\.]+  -> Ofa {avoid}

module Similix-Ova
exports
  lexical syntax
    [A-Za-z0-9\-\!\*\>\?\_\.]+  -> Ova {avoid}

module Similix-OCDs
exports
```

```
lexical syntax
  "defprim-transparent"          -> Key {cons("Key")}
  "defprim-tin"                  -> Key {cons("Key")}
  "defprim-dynamic"              -> Key {cons("Key")}
  "defprim-opaque"               -> Key {cons("Key")}
  "defprim-abort"                -> Key {cons("Key")}
  "defprim-abort-eoi"            -> Key {cons("Key")}
  "defprim"                      -> Key {cons("Key")}

  [0-9]+                         -> Ari {cons("Ari")}
  "*"                            -> Ss {cons("Ss")}
context-free syntax
  "(" Key "(" Ofa V* ")" SchE ")"      -> OCD {cons("KOVE1")}
  "(" Key "(" Ova "." V ")" SchE ")"   -> OCD {cons("KOVE2")}
  "(" Key Ari Ofa SchV ")"             -> OCD {cons("KAOV")}
  "(" Key Ova SchV ")"                 -> OCD {cons("KOV")}
  "(" "defconstr"
      ( "(" C Ss* ")" )+ ")"           -> OCD {cons("Defconstr")}
  "(" "loadt" F ")"                    -> OCD {cons("Loadt2")}
  S                                    -> Ss  {cons("Ss2")}
  E                                    -> SchE {cons("E")}
  V                                    -> SchV {avoid,cons("V4")}
  Ofa                                  -> SchV {avoid,cons("Ofa")}
  Ova                                  -> SchV {cons("Ova")}
```

Appendix B
Similix-Desugar.r

```
module Similix-Desugar
imports Similix OfaArities  Ovas VarRenamer dynamic-rules lib

strategies

  desugar =
    loadfiles
  ; bottomup(try( NamedLetToLetRec
                + InternalDefsToLetRec
                + ListToSeq))
  ; renamevars
  ; topdown(try(lambdalift))
  ; Program(id,map(LetRecToSepFunc);concat,id)
  ; simplerewritings
  ; bottomup(try(MoveLambda))
  ; lookuparis
  ; eta
  ; Program(id,map(MkName),id)

  loadfiles =
    while(FilesToBeLoaded,
            ( Program( map(try(LoadsToLoad);try(LoadLoad) )
                     , id
                     , map(try(LoadsToLoad);try(LoadLoad) ))
            ; IncludeLoad))
  ; repeat(LoadtFile)


  lambdalift =
    rec r({| AddArgs :
             try( LambdaLiftRuleGenerator
                + AddArgs )
           ; all(r)
          |})

  simplerewritings =
    topdown(try( CondToIfThen
               + CondElseToIf
               + IfThenToIf
               + AndToIf
               + OrToIf
```

```
                + LetExpand
                + SeqExpand
                + (  CaseMatchToCond1
                  <+ CaseMatchToCond2)
                + (  CaseConstrToCond1
                  <+ CaseConstrToCond2)))

  lookuparis =
    OfaArities
  ; LookupOvas
  ; Program( map(where(try(UserdefinedConstrs)))
           , map(LookupDefines)
           , id)
  ; topdown(try( LookupAri
               + LookupOva1
               + LookupOva2
               + LookupAriCQ
               + LookupAriC
               + LookupAriS
               + LookupAriP))

  eta =
    topdown( try(  EtaP
                <+ Eta
                <+ Eta2)
           ; try(RemoveEmptyLambda))

   Similix-Desugar = iowrap(desugar)

rules

  NamedLetToLetRec :
    LetP(procname,varsvals,body) ->
    LetRec([(procname,vars,body)],Body([],[App(V(procname),vals)]))
      where <map(?(<id>,_))>varsvals => vars
          ; <map(?(_,<id>))>varsvals => vals

  InternalDefsToLetRec :
    Body([def|defs], exps) ->
    Body([],[LetRec( <map(\Define(a,b,c) -> (a,b,c) \ )>[def|defs]
                   , Body([],exps))])

  ListToSeq :
    Body([],[a,b | rest]) ->
    Body([],[Seq([a,b | rest])])

  LambdaLiftRuleGenerator =
    LetRec(map(MkLambdaLiftRules), id)

  MkLambdaLiftRules :
    (procname, vars, body) ->
    (procname, <conc>(vars,fvs),body)
    where <freevars>Define(procname, vars, body) => fvs
        ; rules(
            AddArgs :
              App(V(procname),args) ->
              App(V(procname),<conc>(args,<map(!V(<id>))>fvs))
          )

  // Collection of free variables from an expression

  // Note: produces list of variable *names* (strings) withouth V(_)

  freevars =
    collect-exc(\ V(x) -> [x] \, FreeVars)

  FreeVars(fv) :
    Define(procname, vars, body) -> <diff> (<fv> body, vars)

  FreeVars(fv) :
    Let(bindings, body) ->
    <diff>(<fv> body, <map(?(<id>,_))> bindings)

  FreeVars(fv) :
    Lambda(vars, body) ->
    <diff>(<fv> body, vars)

  FreeVars(fv) :
    App(_, e) -> <fv> e


  LetRecToSepFunc :
    Define(name,args,body) ->
    <conc>(extradefines,[Define(name,args,body2)])
      where < collect-all(
                ?LetRec(<map(\ (a,b,c) -> Define(a,b,c) \ )>,_))
            ; concat>body => extradefines
          ; <topdown(try(
                ?LetRec(_,Body([],[<id>]))))>body => body2


  UserdefinedConstrs =
    ?Loadt(filename);
      where ( <ParseSimilix;?OCDs(<id>)>filename => contents
            ; filter(KAOV(id,id,id,id) + KOVE1(id,id,id,id))
            ; map( (\KAOV(a,b,c,d) -> (c,<string-to-int>b) \ )
                 + (\KOVE1(a,b,c,d) -> (b,<length>c) \ ) )
                        => lookuplist

            ; list(MakeOfaRules)
            ; < filter(?Defconstr(<id>))
              ; concat> contents => temp
            ; map(?(_,<length;dec;upto <+ []>)) => indices
            ; < zip(MakeSelNames)
              ; concat
              ; list(MakeSelRules)>(<map(AddSelNames)>temp,indices)

            ; <map(\ (a,b) -> (a,<length>b) \ )>temp
            ; list(MakeDefconstrRules)
            ; list(MakeConstrTestRules))
            ; < filter( KOV(id,id,id)
                     + KOVE2(id,id,id,id))
              ; map(MkOvaRules)>contents

  MkOvaRules =
    ?KOV(key,name,def)
    ; rules( LookupOva1 : V(name) -> Ova(name)
             LookupOva2 : P(name) -> Ova(name))

  MkOvaRules =
    ?KOVE2(key,name,var,def)
    ; rules( LookupOva1 : V(name) -> Ova(name)
             LookupOva2 : P(name) -> Ova(name))

  // Parse a similix file and produce its abstract syntax tree

  ParseSimilix :
    filename -> contents
    where new-file => newfile1
        ; new-file => newfile2
        ; <parse-similix>(<un-double-quote>filename, newfile1)
        ; <call>("implode-asfix", ["-i", newfile1, "-o", newfile2])
        ; <ReadFromFile> newfile2 => contents
        ; <call>("rm",["-f",newfile1,newfile2])

  // Call sglr to parse a similix file
  // Note that the .tbl is expected in the current directory

  parse-similix =
    ?(file1, file2);
    <call>("sglr", ["-2","-p","Similix.tbl","-i",file1,"-o",file2])


  AddSelNames :
    (a,b) ->
    <map(\ belem -> (a,belem) \ )>b

  MakeSelNames :
    (name,indices) ->
    <zip(try(ConcSelNames))>(name,indices)

  ConcSelNames :
    ((name,"*"),index) ->
      <concat-strings>[name,".",<int-to-string>index]

  ConcSelNames :
    ((name,Ss2(orig)),index) ->
    orig


  MakeDefconstrRules :
    (name,ari) ->
    (name,ari)
              where rules(LookupAriC : V(name) -> C([name,ari]))

  MakeConstrTestRules :
    (name,ari) ->
    (name,ari)
              where rules(LookupAriCQ : ConsQ(name) ->
                                        ConsQ([name,1]))

  MakeSelRules :
    name ->
    name
             where rules(LookupAriS : V(name) -> S([name,1]))

  MakeOfaRules :
    (name,ari) ->
    (name,ari)
              where rules(LookupAri : V(name) -> Ofa([name,ari]))

   OfaArities =
    where ( <ReadFromFile;?Aris(<id>)>
               "Ofas/OfaArities.aexp" => ofas
          ; map(\ Ari(a,b) -> (a,<string-to-int>b) \ )
          ; map(\ (a,b) -> (a,b)
              where rules( LookupAri  :
                             V(a) ->
                             Ofa([a,b])) \ )
          ; filter(where(?(<IsConstructorTest>,<id>)))
          ; map( \ (a,b) -> (<split-before;Fst>(a,"?"),b) \ )
          ; map( \ (a,b) -> (a,b)
              where rules( LookupAri :
                             ConsQ(a) ->
                             Ofa([<conc-strings>(a,"?"),b])) \ ))

  IsConstructorTest = explode-string;last;?63

  LookupOvas =
    where ( <ReadFromFile;?Ovas(<id>)> "Ovas/Ovas.aexp"
          ; map(\Ova(x) -> x
              where rules(LookupOva1 : V(x) -> Ova(x)
                          LookupOva2 : P(x) -> Ova(x)) \ ))

  LookupDefines =
    ?Define(a,b,c);
    where (rules(LookupAriP : V(a) -> P([a,<length>b])))

  EtaP :
     P([name,0]) ->
     App(P(name),[])

  Eta :
    App(constr#([[name,ari]]),args) ->
    App(constr#([name]),args)
      where <Is-Eta>constr
```

```
        ; <eq>(ari,<length>args)

Eta :
  App(constr#([[name,ari]]),args) ->
  Lambda( newargs
        , Body( []
              , [App( constr#([name])
                    , <conc>(args,newargs1))]))
    where <Is-Eta>constr
        ; not(<eq>(ari,<length>args))
        ; <subt;dec;upto;map(new)>(ari,<length>args) => newargs
        ; map(\ x -> V(x) \ ) => newargs1

Eta :
  Lambda(a,Body([],[Lambda(b,Body([],[App(constr#([c]),d)]))])) ->
  Lambda(<conc>(a,b),Body([],[App(constr#([c]),d)]))
    where <Is-Eta>c

Eta2 :
  constr#([[name,0]]) ->
  constr#([name])
    where <Is-Eta>constr

Eta2 :
  constr#([[name,ari]]) ->
  Lambda(args,Body([],[App(constr#([name]),args1)]))
    where <Is-Eta>constr
        ; <dec;upto;map(new)>ari => args
        ; map(\ x -> V(x) \ ) => args1

Is-Eta = ?"Ofa"
       + ?"C"
       + ?"ConsQ"
       + ?"S"
       + ?"P"


IfThenToIf :
  IfThen(e1,e2) ->
  If(e1,e2,V("dummy"))


CondToIfThen :
  Cond([(a,b)]) ->
  IfThen(a,b)

CondToIfThen :
  Cond([(a,b),(c,d) | conds]) ->
  If(a,b,Cond([(c,d)| conds]))


CondElseToIf :
  CondElse([(a,b)],else) ->
  If(a,b,else)
```

```
CondElseToIf :
  CondElse([(a,b), (c,d) | conds],else) ->
  If(a,b,CondElse([(c,d) | conds],else))



AndToIf :
  And([a]) ->
  a

AndToIf :
  And([a,b | rest]) ->
  If(a,And([b | rest]),Bool("#f"))



OrToIf :
  Or([a]) ->
  a

OrToIf :
  Or([a,b | rest]) ->
  If(a,Bool("#t"),Or([b|rest]))


LetExpand :
  Let([(a,b),(c,d) | rest],e) ->
  Let([(a,b)],Body([],[Let([(c,d) | rest],e)]))

LetExpand :
  LetStar([(a,b),(c,d) | rest],e) ->
  Let([(a,b)],Body([],[Let([(c,d) | rest],e)]))


SeqExpand :
  Seq([a,b,c | rest]) ->
  Seq([a,Seq([b,c | rest])])


CaseMatchToCond1 :
  If( App(p,[var,q])
    , r
    , CaseMatch(var,[(WiC(a),b)])) ->
  If( App(p,[var,q])
    , r
    , b)

CaseMatchToCond1 :
  If( App(p,[var,q])
    , r
    , CaseMatch(var,[(a,b)])) ->
  If( App(p,[var,q])
    , r
    , IfThen(App(Ofa("equal?"),[var,a]),b))
    where not(<?WiC(_)>a)

CaseMatchToCond1 :
  If( App(p,[var,q])
    , r
    , CaseMatch(var,[(a,b), (c,d) | rest])) ->
  If( App(p,[var,q])
    , r
    , If( App(Ofa("equal?"),[var,a])
        , b
        , CaseMatch(var,[(c,d) | rest])))

CaseMatchToCond2 :
  CaseMatch(var,[(a,b)]) ->
  IfThen(App(Ofa("equal?"),[var,a]),b)

CaseMatchToCond2 :
  CaseMatch(var,[(a,b), (c,d)|rest]) ->
  Let([(V(z),var)],Body([],If( App(Ofa("equal?"),[V(z),a])
                             , b
                             , CaseMatch(V(z),[(c,d) | rest]))))
    where new => z



CaseConstrToCond1 :
  If( App(p,[var,q])
    , r
    , CaseConstr(var,[(WiC(a),b)])) ->
  If( App(p,[var,q])
    , r
    , b)

CaseConstrToCond1 :
  If( App(p,[var,q])
    , r
    , CaseConstr(var,[(a,b)])) ->
  If( App(p,[var,q])
    , r
    , IfThen(App(Ofa("equal?"),[var,a]),b))
    where not(<?WiC(_)>a)

CaseConstrToCond1 :
  If( App(p,[var,q])
    , r
    , CaseConstr(var,[(a,b), (c,d) | rest])) ->
  If( App(p,[var,q])
    , r
    , If( App(Ofa("equal?"),[var,a])
        , b
        , CaseConstr(var,[(c,d) | rest])))

CaseConstrToCond2 :
  CaseConstr(var,[(a,b)]) ->
  IfThen(App(Ofa("equal?"),[var,a]),b)

CaseConstrToCond2 :
  CaseConstr(var,[(a,b), (c,d) |rest]) ->
  Let([(V(z),var)],Body([],If( App(Ofa("equal?"),[V(z),a])
                             , b
                             , CaseConstr(V(z),[(c,d) | rest]))))
    where new => z


FilesToBeLoaded =
  ?Program(a,_,c);
  where(<conc;filter(Load(id) + Loads(id))>(a,c) => [_|_])

LoadsToLoad :
  Loads(filename) ->
  Load(filename)

LoadLoad =
  Load(ParseSimilix)


IncludeLoad :
  Program(a,b,c) ->
  Program(<concat>[listofloadts,a2,c2],<conc>(b,b2),[])
    where <conc;filter(Load(id))>(a,c) => listofloads
        ; <conc;filter(Loadt(id))>(a,c) => listofloadts
        ; <map(?Load(Program(<id>,_,_)));concat>listofloads => a2
        ; <map(?Load(Program(_,_,<id>)));concat>listofloads => c2
        ; <map(?Load(Program(_,<id>,_)));concat>listofloads => b2

LoadtFile :
  Program(a,b,c) ->
  Program(res,b,[])
    where <conc;map(?Loadt(<id>))>(a,c) => listofloadts
        ; < map( ParseSimilix
               ; collect(?Loadt2(<id>)))
          ; concat>listofloadts => listofloadt2s
        ; < union; map( \ x -> Loadt(x) \ )>
              (listofloadt2s,listofloadts) => res
        ; not(<eq>(res,<conc>(a,c)))


MkName :
  Define(a,b,c) -> Define(Name(a,""),b,c)

MoveLambda :
  App(Lambda(args,body),vals) ->
  App(Lambda(args2,Body([],[App(Lambda(args1,body),vals)])),[])
    where <Split>(args,[],vals) => (args1,args2)
        ; not(<eq>(args2,[]))

Split :
  ([arg|args],args1,[val|vals]) ->
  <Split>(args,<conc>(args1,[arg]),vals)
Split :
  (args2,args1,[]) ->
  (args1,args2)

MoveLambda :
 Define( name
       , args
       , Body([],[App(Lambda(largs,Body([],body)),[])])) ->
 Define(name,<conc>(args,largs),Body([],body))

MoveLambda :
 If(cond-part,then-part,else-part) ->
 App(Lambda( [arg|args]
          , Body([],[If( <RmEmptyLambda>cond-part
                       , <RmEmptyLambda>then-part
                       , <RmEmptyLambda>else-part)])),[])
    where <GetLambdaArgs>[ cond-part
                         , then-part
                         , else-part] => [arg|args]

MoveLambda :
 Seq([a,b]) ->
 App(Lambda( [arg|args]
          , Body([],[Seq( [<RmEmptyLambda>a
                         , <RmEmptyLambda>b])])),[])
    where <GetLambdaArgs>[a,b] => [arg|args]

MoveLambda :
   Let([(v,e)],Body([],[body])) ->
   App(Lambda( [arg|args]
            , Body([],[Let( [(v,<RmEmptyLambda>e)]
                          ,Body( []
                               , [<RmEmptyLambda>body]))])),[])
    where <GetLambdaArgs>[e,body] => [arg|args]

MoveLambda :
   App(App(Lambda([a|as],body),[]),args) ->
   < try(MoveLambda)
   ; try(RmEmptyLambda)>App(Lambda([a|as],body),args)

MoveLambda :
   App(Lambda( args1
            , Body( []
                  , [App( Lambda(args2,body2)
                         , vals2)])),vals1) ->
   <MoveLambda;try(RmEmptyLambda)>
   App(Lambda( <conc>(args1,args2),body2)
            , <conc>(vals1,vals2))

RmEmptyLambda = try(?App(Lambda(_,Body([],[<id>])),[]))

RemoveEmptyLambda = ?Lambda([],Body([],[<id>]))
```

```
GetLambdaArgs = filter(?App(Lambda(<id>,_),[]));concat
```

Appendix C

Similix-BTA.r

```
module Similix-BTA
imports Similix Preparator Annotator NameCreator Similix-Annotations
        dynamic-rules lib

strategies
  startbta =
    prepare
  ; where( ?Program(<map(OfaOvaRules)>,_,_))
  ; where( <table-create>"defs"
         ; <table-create>"pending"
         ; <table-put>("pending", "pending", [])
         )
  ; where( ?Program( _
                   , < fetch(LookupBTAfunction)
                     ; filter(ToBTA)
                     ; ?[<id>]>
                   ,  _)
         ; bta
         )
  ; Program( id
           , map(AnnotateDefinition <+ ![<id>]); concat
           , id) => p
  ; (
     ( <eq> (<collect-annos(Rec)>p,[]); !p )
     <+ bottomup(try(AnnotateRec))
    )


  bta =
  ; recursion-check
  ; try( ?Define( name@Name(realname,"")
                , (namedargs,valueargs)
                , body)
       ; btastrategy
       ; ?Define( _, _, body2 )
       ; <?Body([],[<GetAnnotation>])>body2 => anno
       ; <map(Annotate;RmVals)>valueargs => annos
       ; <FinishBTA>Define( name
                          , (namedargs,valueargs)
                          , body2) => btaddefine
       ; where( !(btaddefine,body)
      ; updatetables)
       ; rules(
           ProcAnno :
             App(P(realname),annos) -> anno

           AnnotateDefinition :
     Define(name, argsa, body) ->
             <table-get>("defs", (name,argsa) )
              )
       )

  recursion-check =
    ?Define( name@Name(realname,"")
           , (namedargs,valueargs)
           , body ) => define
  ; !App( P(realname)
        , <zip(MoveAnnos)>( namedargs
                          , <map(Annotate)>valueargs)) => app
  ;  ( alreadypending; !define)
    <+ !BTAd(Define("name","args",Body([],["expr"{Rec}])))

  updatetables =
    ?( btaddefine@BTAd(Define(name@Name(realname,_),namedargs,body2))
     , body)
  ; <map(RmAnnos)>namedargs => args
  ; ( <table-get> ( "defs", (Name(realname,""),args))
   <+ ![Define(Name(realname,""),args,body)]) => list
  ;  <table-put> ( "defs", (Name(realname,""),args)
                         , <conc>(list, [btaddefine]) )

  alreadypending =
    ?app
  ; <table-get>("pending","pending") => pendinglist
  ; <collect(\x -> x where <eq>(x,app) \ )>
        pendinglist => alreadypending
  ; <eq>(alreadypending,[])
  ; <table-put>("pending","pending",[app | pendinglist])

  btastrategy =
    rec distributeannos
      ({|  AnnotateLetBoundVars,
           AnnotateVars :
           try( Let([(id,distributeannos)],id)
              + App(Lambda(id,id),distributeannos))
         ; try( LetVarRuleGenerator
              + VarRuleGenerator
              + AnnotateLetBoundVars
              + AnnotateVars
              + LambdaVarRuleGenerator
              + MkConstantsStatic
              )
         ; ( Let([(distributeannos,id)],distributeannos)
          <+ App(Lambda(id,distributeannos),id)
          <+ Define(id,id,distributeannos)
```

```
          <+ all(distributeannos))
         ; bottomupannotate
        |})

  bottomupannotate =
    try( ( AnnotateLambda
         + AnnotateConsQ
         + AnnotateSel
         + AnnotateConstr)
      <+ ( AnnotateIfRec
        <+ AnnotateIf)
      <+ AnnotateApp
       + AnnotateLet
       + AnnotateLis
       + AnnotateSeq
       + AnnotateVar)

  Similix-BTA = iowrap(startbta)

rules

  AnnotateRec :
    x{a} ->
    y{anno}
      where not(<Constant>x)
          ; not(<Variable>x)
          ; <bottomupannotate>x => y{anno}

  Variable = ?V(_) + is-string

  LookupBTAfunction =
    ?Define(Name("bta",""),args,body);
      where ( <?Body(_,[<id>])>body => app
          ; <?App(P(<id>),_);(\x -> Name(x,"") \ )>app => appname
          ; <?App(_,<id>)>app => appargs
          ; rules(ToBTA :
                Define(appname,oldargs,funcbody) ->
                Define(appname,(oldargs,appargs),funcbody)
                  where <eq>(<length>appargs,<length>oldargs)))

  VarRuleGenerator =
    ?Define(name,(namedargs,valueargs),body);
      where( <map(Annotate)>valueargs => annotatedvalueargs
           ; <zip(MkVarAnnotationRules)>
                (namedargs,annotatedvalueargs))

  MkVarAnnotationRules =
    ?(var,val{anno});
      where (rules(AnnotateVars : V(var) -> V(var){anno}))

  LetVarRuleGenerator :
    Let([(var,val{anno})],body) ->
    Let([(var{anno},val{anno})],body)
      where <is-string>var
```

```
        ; rules(AnnotateLetBoundVars : V(var) -> V(var){anno})

LambdaVarRuleGenerator :
 App(Lambda(vars,body),vals) ->
 App(Lambda(annvars,body),vals)
   where <zip(MoveAnnos)>(vars,vals) => annvars
       ; <zip(MkVarAnnotationRules)>(vars,vals)

AnnotateApp :
  app@App(x,args) ->
  app{<GetAnnos>app}
    where not(<?P(_)>x)

AnnotateApp :
  app@App(P(x),args) ->
  App(P(y),args){<GetAnnos>app}
    where (<?Name(_,_)>x <+ <BtaName>app <+ !x) => y

AnnotateConsQ :
  App(ConsQ(x),[arg]) ->
  App(ConsQ(x),[arg]){anno}
    where <GetAnnotation;Hd>arg => anno

AnnotateConstr :
  App(C(x),args) ->
  App(C(x),args){annargs}
    where <conc>([St],<map(RmVals1 <+ RmVals)>args) => annargs

RmVals1 :
  x{annos} -> Annotation(annos)
    where <is-list>annos

AnnotateIf :
  If(a{St},b{anno},c{anno}) ->
  If(a{St},b{anno},c{anno}){anno}

AnnotateIf :
  If(a{condanno},b{banno},c{canno}) ->
  If(a{condanno},b{banno},c{canno}){Dy}
    where (not(<eq>(condanno,St)) + not(<eq>(banno,canno)))

AnnotateIfRec :
  If(a{St},b{annob},c{annoc}) ->
  If(a{St},b{annob},c{annoc}){anno}
    where <filter(not(Rec))>[annob,annoc] => [anno]

AnnotateLambda :
  App(Lambda(a,Body([],[b{anno}])),args) ->
  App(Lambda(a,Body([],[b{anno}])),args){anno}

AnnotateLet :
  Let([b],Body([],[x{anno}])) ->
  Let([b],Body([],[x{anno}])){anno}
```

```
AnnotateLis :
  Quote(Lis1(x){anno}) ->
  Quote(Lis1(x)){anno}

AnnotateLis :
  Lis1(x) ->
  Lis1(x){LisAnno(anno)}
    where <map(RmVals)>x => anno

AnnotateLis :
  Quote(Lis2(x,y){anno}) ->
  Quote(Lis2(x,y)){anno}

AnnotateLis :
  Lis2(x,y) ->
  Lis2(x,y){LisAnno(anno)}
    where <map(RmVals)>[x,y] => anno

AnnotateSel :
  App(S(x),[arg{arganno}]) ->
  App(S(x),[arg]){anno}
    where <index>(<LookupS;inc>S(x),arganno) => anno

AnnotateSeq :
  Seq([a,b{anno}]) ->
  Seq([a,b{anno}]){anno}

AnnotateVar :
  V(x){} -> V(x){Dy}

MoveAnnos :
  (a,b{anno}) -> a{anno}

FinishBTA :
  Define(Name(name,""),(args1,args2),body) ->
  BTAd(Define(name2,args3,body))
    where <zip(MoveAnnos)>(args1,<map(Annotate)>args2) => args3
        ; <map(try(MkBTAName));concat-strings>args3 => pt2
        ; !Name(name,<conc-strings>("-a",pt2)) => name2

collect-annos(x) :
  a{b} -> <conc>(<collect-annos(x)>a,<collect(x)>b)

OfaOvaRules =
  ?Loadt(filename) ->
    where ( <ParseSimilix>filename => contents
          ; < ?OCDs(<id>)
            ; filter( KOVE1(id,id,id,id)
                    + KOVE2(id,id,id,id)
                    + KAOV(id,id,id,id)
                    + KOV(id,id,id) )
            ; map( (\KOVE1(key,ofa,vars,_) ->
                     (key,ofa,<length>vars) \ )
                 + (\KAOV(key,ari,ofa,_) ->
                    (key,ofa,<string-to-int>ari) \ )
                + (\KOV(key,ova,_) ->
                    (key,ova) \ )
                + (\KOVE2(key,ova,_,_) ->
                    (key,ova) \ ))>contents => ofasovas
          ; <filter((id,id,id))>ofasovas => ofas
          ; <map(try(MkOfaRules))>ofas
          ; <filter((id,id))>ofasovas => ovas
          ; <map(try(MkOvaRules))>ovas

MkOfaRules =
  ?(key,ofa,ari)
  ; < ?"defprim-dynamic"
    + ?"defprim-opaque"
    + ?"defprim-abort"
    + ?"defprim-abort-eoi">key
  ; rules(GetSpecificAnnos : App(Ofa(ofa),args) -> Dy
           where <eq>(<length>args,ari))

MkOvaRules =
  ?(key,ova)
  ; < ?"defprim-dynamic"
    + ?"defprim-opaque"
    + ?"defprim-abort"
    + ?"defprim-abort-eoi">key
  ; rules(GetSpecificAnnos : App(Ova(ova),args) -> Dy)


FixName :
  App(P(Name(x,_)),args) ->
  App(P(x),args)

GetAnnos =
  try(FixName)
; ( RecAnno
 <+ ProcAnno'
 <+ GetSpecificAnnos
 <+ GetGeneralAnnos )

ProcAnno' :
  App(P(func),args) -> anno
    where <map(Annotate;RmVals)>args => args'
        ; <ProcAnno>App(P(func),args') => anno

RecAnno :
  App(func,args) -> Rec
    where <filter(?_{Rec})>args => [_|_]

GetGeneralAnnos :
  App(func,args) -> St
    where <eq>(<filter(?_{<eq>(<collect(Dy)>,[])})>args,args)

GetGeneralAnnos :
  App(func,args) -> Dy
```

```
    where not(<eq>(<filter(?_{St})>args,args))
        ; <eq>(<filter(?_{_})>args,args)

GetSpecificAnnos :
  App(P(x),args) ->
  annos
    where <LookupFunction>(Name(x,""),<length>args)
             => (args2,body)
        ; <bta>Define(Name(x,""),(args2,<map(Annotate)>args),body)
             => BTAd(Define(_,_,Body(_,[body])))
        ; <GetAnnotation>body => annos

GetSpecificAnnos :
  App(Ofa("assoc"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("call-with-input-file"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("call-with-output-file"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("close-input-port"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("close-output-port"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("current-input-port"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("current-output-port"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("input-port?"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("list-ref"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("member"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("open-input-file"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("open-output-file"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("output-port?"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("vector-ref"),args) -> Dy
```

```
GetSpecificAnnos :
  App(Ova("display"),args) -> Dy

GetSpecificAnnos :
  App(Ova("make-vector"),args) -> Dy

GetSpecificAnnos :
  App(Ova("newline"),args) -> Dy

GetSpecificAnnos :
  App(Ova("peek-char"),args) -> Dy

GetSpecificAnnos :
  App(Ova("read"),args) -> Dy

GetSpecificAnnos :
  App(Ova("read-char"),args) -> Dy

GetSpecificAnnos :
  App(Ova("write"),args) -> Dy

GetSpecificAnnos :
  App(Ova("write-char"),args) -> Dy

GetSpecificAnnos :
  App(Ova("_sim-error"),args) -> Dy

GetSpecificAnnos :
  App(Ofa("reverse"),[a1]) ->
  LisAnno(annos)
    where <GetAnnotation;?LisAnno(<reverse>)>a1 => annos

GetSpecificAnnos :
  App(Ofa("cons"),[a1,a2]) ->
  LisAnno([anno1|anno2])
    where <GetAnnotation>a1 => anno1
        ; <GetAnnotation;?LisAnno(<id>)>a2 => anno2

GetSpecificAnnos :
  App(Ofa("cons"),[a1,a2]) ->
  LisAnno([anno1,anno2])
    where <GetAnnotation>a1 => anno1
        ; <GetAnnotation>a2 => anno2
        ; not(<?LisAnno(_)>anno2)

GetSpecificAnnos :
  App(Ofa("car"),[x]) -> res
    where <GetAnnotation;?LisAnno(<Hd>)>x => res

GetSpecificAnnos :
  App(Ofa("cdr"),[x]) -> LisAnno(res)
    where <GetAnnotation;?LisAnno(<Tl>)>x => res

GetSpecificAnnos :
  App(Ofa("length"),[x]) -> St
    where not(<eq>(<GetAnnotation;last>x,Dy))

GetSpecificAnnos :
  App(Ofa("list->string"),[x]) -> St
    where <GetAnnotation;collect(Dy)>x => []

GetSpecificAnnos :
  App(Ofa("list->vector"),[a{x}]) -> x

GetSpecificAnnos :
  App(Ofa("vector->list"),[a{x}]) -> x

GetSpecificAnnos :
  App(Ofa("list-ref"),[list{x},num{St}]) -> St
    where <collect(Dy)>x => []

GetSpecificAnnos :
  App(Ofa("list?"),[x{y}]) -> St
    where not(<eq>(y,Dy))

GetSpecificAnnos :
  App(Ofa("null?"),[x{y}]) -> St
    where <collect(Dy)>y => []

GetSpecificAnnos :
  App(Ofa("pair?"),[x{y}]) -> St
     where not(<eq>(y,Dy))

GetSpecificAnnos :
  App(Ofa("vector-length"),[x]) -> St
    where not(<eq>(<GetAnnotation;last>x,Dy))

GetSpecificAnnos :
  App(Ova("list"),x) -> LisAnno(<map(GetAnnotation)>x)

GetSpecificAnnos :
  App(Ova("max"),x) -> St
    where <GetAnnotation;collect(Dy)>x => []

GetSpecificAnnos :
  App(Ova("min"),x) -> St
    where <GetAnnotation;collect(Dy)>x => []

GetSpecificAnnos :
  App(Ova("append"),x) ->
  LisAnno(<map(GetAnnotation;?LisAnno(<id>) <+ ![Dy]);concat>x)
```

Appendix D
Similix-BTI.r

```
module Similix-BTI
imports Similix Similix-Annotations lib

strategies
  startbti = topdown(try(BTICalc))

  Similix-BTI = iowrap(startbti)


rules

  BTICalc :
    App(Ova(x),args){Dy} ->
    App(Ova(x),<conc>([dy|dys],[App(Ova(x),[st|sts]){St}])){Dy}
      where (<eq>(x,"*") + <eq>(x,"+"))
          ; <collect(?_{Dy})>args => [dy|dys]
          ; <collect(?_{St})>args => [st|sts]

  BTICalc :
    App(Ova(x),[arg|args]){Dy} ->
    App(Ova(x),<conc>( [arg|dys]
                     , [App(Ova(<inv>x),[st|sts]){St}])){Dy}
      where (<eq>(x,"-") + <eq>(x,"/"))
          ; <get-annotations>arg => [Dy]
          ; <collect(?_{Dy})>args => dys
          ; <collect(?_{St})>args => [st|sts]

  BTICalc :
    App(Ova(x),[arg|args]){Dy} ->
    App(Ova(x),<conc>([App(Ova(x),[arg{St},App(Ova(<inv>x)
                               ,[st|sts]){St}]){St}],dys))
      where (<eq>(x,"-") + <eq>(x,"/"))
          ; <get-annotations>arg => [St]
          ; <collect(?_{Dy})>args => dys
          ; <collect(?_{St})>args => [st|sts]

  inv : "-" -> "+"
  inv : "/" -> "*"
```

Appendix E
Similix-PE.r

```
module Similix-PE
imports Similix Similix-Annotations Preparator Annotator NameCreator
        dynamic-rules lib

strategies
  startpe =
    topdown( try( NumToNum
                + CharToChar
                + StrToStr))
  ; prepare
  ; where( <table-create>"defs" )
  ; where( <table-create>"pending" )
  ; where( ?Program(<map(OfaOvaRules)>,_,_) )
  ; where( ?Program( _
                   , < fetch(LookupBTAfunction)
                     ; filter(ToPE)
                     ; ?[BTAd(<id>)]>
                   , _)
         ; pe)
  ; Program(id
           , map(FillDefinition <+ ![<id>])
             ; concat
             ; filter(not(Superfluous))
           , id)
  ; topdown( try( NumFromNum
                + CharFromChar
                + StrFromStr))

  pe =
    ?Define(name,(namedargs,valueargs),body) => define
  ; where(updatependingtable)
  ; pestrategy
  ; ?Define( name
           , (_, valueargs)
           , body2@Body(_,[innerbody2])) => define2
  ; <map(MkStandard)>valueargs => args'
  ; where(!(define2,body);updatedefstable)
  ; !innerbody2
  ; rules( FillProc :
             App(P(name), args') -> innerbody2
           FillDefinition :
             BTAd(Define(name,namedargs,body)) ->
             <table-get>("defs",(name,namedargs))
          )

  updatependingtable =
    ?Define(name,(namedargs,valueargs),body)
  ; <PeName>App( P(name)
               , <map(topdown(try( NumFromNum
                                 + CharFromChar
                                 + StrFromStr)))>valueargs)
          => pename
  ; <table-put>("pending",pename,pename)


  updatedefstable =
    ?(Define(name,(namedargs,valueargs),body2),body)
  ;  ( <table-get>("defs",(name,namedargs))
    <+ ![BTAd(Define(name,namedargs,body))]) => list
  ; <table-put>( "defs"
               , (name,namedargs)
               , <concat>[ [<Hd>list]
                         , [<FinishPE>Define(name
                              , (namedargs,valueargs)
                              , body2)]
                         , <Tl>list])

  pestrategy =
    rec distributevals
      ({| FillVars,
          AddInfo,
          AddInfoRec,
          FillLetBoundVars,
          AddInfoLetBoundVars :
            try(UserDefinedPrim)
          ; try( Let([(id,distributevals)],id)
               + App(Lambda(id,id),distributevals)
               + If(distributevals,id,id))
          ; try( VarRuleGenerator
               + AddInfo
               + AddInfoRec
               + FillVars
               + LetVarRuleGenerator
               + AddInfoLetBoundVars
               + LambdaVarRuleGenerator
               + FillLetBoundVars
               )
          ;  ( Let([(distributevals,id)],distributevals)
            <+ App(Lambda(id,distributevals),id)
            <+ Define(id,id,distributevals)
            <+ IfTrue(distributevals)
            <+ IfFalse(distributevals)
            <+ Info(id,id)
            <+  all(distributevals))
          ; bottomupevaluate
       |})

    bottomupevaluate =
      try( ( RemoveIdLets
           + RemoveLambdas)
         <+ ( ( FillApp
              <+ RenameApp)
           + FillIf)
         <+ RmAnnos)

    Similix-PE = iowrap(startpe)

rules

  IfTrue(s) : If(Bool("#t"),thenpart,elsepart){annot} ->
              If(Bool("#t"),<s>thenpart,0){annot}

  IfFalse(s) : If(Bool("#f"),thenpart,elsepart){annof} ->
               If(Bool("#f"),0,<s>elsepart){annof}

  RmInfos = ?Info(<id>,_)

  Superfluous = ?BTAd(_)

  MkStandard = <Constant>x
  MkStandard : x -> V("x")
    where not(<Constant>x)

  NumToNum : Num(x){} -> Num(<string-to-num>x){}

  NumToNum : Num(x){y} -> Num(<string-to-num>x){y}

  NumFromNum = ?Num(<num-to-string>);!Num(<id>)

  CharToChar : Char(c){} -> Char(<un-single-quote>c){}

  CharToChar : Char(c){y} -> Char(<un-single-quote>c){y}

  CharFromChar = ?Char(<single-quote>);!Char(<id>)

  StrToStr :
   Str(x){} -> Str(<un-double-quote>x){}

  StrToStr :
   Str(x){y} -> Str(<un-double-quote>x){y}

  StrFromStr = ?Str(<double-quote>);!Str(<id>)

  LookupBTAfunction :
    Define(Name("bta",""),args,body) ->
    Define(Name("bta",""),args,body)
      where <?Body(_,[<id>])>body => app
          ; <?App(_,<id>)>app => appargs
```

```
        ; <BtaName>App( <?App(<id>,_)>app
                      , <map(Annotate)>appargs)
               => appname
        ; rules(ToPE :
                  BTAd(Define(appname,oldargs,funcbody)) ->
                  BTAd(Define(appname,(oldargs,appargs),funcbody))
                    where <eq>(<length>appargs,<length>oldargs))

LambdaVarRuleGenerator :
  App(Lambda(vars,body),vals) ->
  App(Lambda(vars,body),vals)
    where < zip(ExistsStatic)
          ; filter(not(is-string))
          ; map(MkAddInfoRules)>(vars,vals)
        ; < zip(AllStatic)
          ; filter(not(is-string))
          ; map(MkFillVarRules)>(vars,vals)

VarRuleGenerator :
  Define(name,(namedargs,valueargs),body) ->
  Define(name,(namedargs,valueargs),body)
    where < zip(ExistsStatic)
          ; filter(not(is-string))
          ; map(MkAddInfoRules)>(namedargs,valueargs)
        ; < zip(AllStatic)
          ; filter(not(is-string))
          ; map(MkFillVarRules)>(namedargs,valueargs)

AllStatic :
  (x{annos},b) ->
  (V(x{}){annos},b)
    where <collect(Dy)>annos => []

AllStatic :
  (x{annos},b) -> x
    where not(<eq>(<collect(Dy)>annos,[]))

ExistsStatic :
  (x{annos},b) ->
  (V(x{}){annos},b)
    where not(<eq>(<collect(St)>annos,[]))
        ; not(<eq>(<collect(Dy)>annos,[]))

ExistsStatic :
  (x{annos},b) -> x
    where ( <eq>(<collect(St)>annos,[])
          + <eq>(<collect(Dy)>annos,[]))

MkFillVarRules :
  (var,val) ->
  (var,val)
    where rules(FillVars : var -> val)

MkAddInfoRules :
  (var,Info(var2,val)) ->
  (var,Info(var2,val))
    where rules(AddInfoRec : var -> Info(var,val))

MkAddInfoRules :
  (var,val) ->
  (var,val)
    where not(<?Info(_,_)>val)
        ; rules(AddInfo : var -> Info(var,val))

LetVarRuleGenerator :
  Let([(var{anno},val)],body) ->
  Let([(val,val)],body)
    where <AllStatic>(var{anno},val) => (var',val)
        ; rules(FillLetBoundVars :   var' -> val)

LetVarRuleGenerator :
  Let([(var{anno},val)],body) ->
  Let([(Info(var{},value),val)],body)
    where <try(?Info(_,<id>))>val => value
        ; <ExistsStatic>(var{anno},value) => (var',value)
        ; rules(AddInfoLetBoundVars : var' -> Info(var'{},value))

RemoveIdLets :
  Let([(a,a)],Body([],[body])) -> body

RemoveLambdas = RemoveLambdas1;try(RemoveLambdas2)

RemoveLambdas1 :
  App(Lambda(args,body),vals) ->
  App(Lambda(args',body),vals')
    where <filter(NoDys)>args => args'
        ; <filter(NoDys)>vals => vals'

RemoveLambdas2 :
  App(Lambda([],Body([],[body])),[]) ->
  body

NoDys : x{anno} -> x{} where <collect(Dy)>anno => [_|_]

FillApp :
  App(x,y) ->
  seq
    where <filter(?Seq(_))>y => [a | b]
        ; <map(try(?Seq([_,<id>])))>y => y2
        ; < map(try(?Seq([<id>,_])))
          ; foldr( <GetRes>App(x,y2)
                 , ( \ (a,b) -> Seq([a,b]) \ ))>y
                        => seq

FillApp :
  App(x,y){annos} ->
  <GetRes>App(x,y){}
    where <collect(Dy)>annos => []
```

```
        ; <filter(?Seq(_))>y => []

 FillApp :
   App(P(x),y){annos} ->
   App(P(z),y2){}
     where <collect(Dy)>annos => [_|_]
         ; <PeName>App(P(x),<map(topdown(try( NumFromNum
                                            + CharFromChar
                                            + StrFromStr)))>y) => z
         ; <filter(not(Constant))>y => y2
         ; <table-get>("pending",z) => z

 FillApp :
   App(P(x),y){annos} ->
   App(P(z),y2){}
     where <collect(Dy)>annos => [_|_]
         ; <PeName>App(P(x),<map(topdown(try( NumFromNum
                                            + CharFromChar
                                            + StrFromStr)))>y) => z
         ; not(<eq>(<table-get>("pending",z),z))
         ; <filter(not(Constant))>y => y2
         ; <LookupFunction>(x,<length>y) => (namedargs,body)
         ; <endingrecursion>([x],body)
         ; <pe>Define(x,(namedargs,y),body)

 RenameApp :
   App(P(Name(x,_)),y){annos} ->
   App(P(Name(x,"")),y){}

 endingrecursion =
   endingrecursion1
<+ endingrecursion2
<+ endingrecursion3
<+ endingrecursion4

 endingrecursion1 =
   ?(names,body)
   ; <collect-all(App(P(id),id))>body => []

 endingrecursion2 =
   ?(names,body)
   ; <collect-all(
       If(( \ x{anno} -> <eq>(anno,St) \ )
         ,( \ x -> <endingrecursion>(names,x) \ )
         ,id)
     + If(( \ x{anno} -> <eq>(anno,St) \ )
         ,id
         ,( \ x -> <endingrecursion>(names,x) \ )))>body => [_|_]

 endingrecursion3 =
   ?(names,body)
   ; < collect(If(id,id,id))
     ; map( \ If(a,b,c) -> ((names,b),(names,c)) \ )
     ; concat>body => ifs
```

```
  ; <fetch((endingrecursion,endingrecursion))>ifs => [_,_]

endingrecursion4 =
  ?(names,body)
  ; <collect(If(id,id,id))>body => []
  ; <collect(App(P(id),id))>body => apps
  ; <filter(App(P(( \ x -> <collect(\ y ->
                                    <eq>(x,y) \ )>names \ ))
              ,id))>apps => temp
  ; <zip(EmptyOrNothing <+ ![]);concat>(temp,apps) => info
  ; <map(Fst;( \ x -> <conc>([x],names) \ ))>info => names'
  ; <map(LookupFunction;Snd)>info => bodies
  ; <zip(id)>(names',bodies) => newnamesbodies
  ; <fetch(endingrecursion)>newnamesbodies => [_|_]

EmptyOrNothing :
  (App(P([]),args),App(P(x),args)) ->
  [(x,<length>args)]


FillIf :
  If(Bool("#t"),thenpart,_){annos} ->
  thenpart

FillIf :
  If(Bool("#f"),_,elsepart){annos} ->
  elsepart

FinishPE :
  Define(Name(name,btapt),(args1,args2),body) ->
  PEd(Define(name2,args3,body))
    where < map( topdown(try( NumFromNum
                            + CharFromChar
                            + StrFromStr))
               ; MkPEName)
          ; concat-strings>args2 => pt2
        ; !Name(name,<conc-strings>("-pe",pt2)) => name2
        ; < zip(id)
          ; filter((not([]),id))
          ; map(Snd;RmAnnos)>
              ( <map(collect-annos(Dy))>args1
              , args1) => args3

collect-annos(x) : a{b} -> <collect(x)>b

OfaOvaRules =
   ?Loadt(filename);
   where (<ParseSimilix;?OCDs(<id>)>filename => contents
       ;  < filter( KOVE1(id,id,id,id) )
          ; map(try(MkKOVE1Rules))>contents
       ;  < filter( KAOV(id,id,id,id)
                  + KOV(id,id,id))
          ; map(try(MkKOVRules))>contents)
```

```
desugar = KOVE1ToDefine;try(Desugar)

KOVE1ToDefine :
  KOVE1(key,name,vars,E(body)) ->
  [Define(name,vars,Body([],[body]))]

Desugar : [def|defs] -> contents
  where new-file => newfile1
      ; new-file => newfile2
      ; <WriteToTextFile>(newfile1,Program([],[def|defs],[]))
      ; <call>("./Similix-Desugar",
                 [ "-i", newfile1
                 , "-o", newfile2])
      ; <ReadFromFile> newfile2 => contents
      ; <call>("rm",["-f",newfile1,newfile2])

MkKOVE1Rules =
  ?KOVE1(key,name,vars,body);
  where( <Reducible>key;
         rules( UserDefinedPrim :
                App(Ofa(name),args){anno} ->
                <MkStatic>App(Lambda(vars,body2),args)
                  where <collect(Dy)>anno => []
                      ; <desugar>KOVE1(key,name,vars,body)
                          => Program([],[Define(_,_,body2)],[])))

MkKOVRules =
  ?KAOV(key,ari,name,Ova(primi));
  where (<Reducible>key;
         rules( UserDefinedPrim :
                  App(Ofa(name),args){anno} ->
                  App(Ofa(primi),args){anno}
                    where <collect(Dy)>anno => []))

MkKOVRules =
  ?KOV(key,name,Ova(ova));
  where (<Reducible>key;
         rules( UserDefinedPrim :
                  App(Ova(name),args){anno} ->
                  App(Ova(ova),args){anno}
                    where <collect(Dy)>anno => []))


MkStatic :
  App( Lambda(vars,Body([],body)),args) ->
  App( Lambda( <map(MkStatic)>vars
             , Body([],<bottomup(try(MkStatic2))>body))
     , args){St}

MkStatic :
  a -> a{St}
    where <is-string>a

MkStatic2 :
  a -> a{St}
    where < ?V(_)
          + ?App(_,_)
          + ?Let(_,_)
          + ?Lambda(_,_)
          + ?Seq(_)>a

Reducible = ?"defprim"
          + ?"defprim-transparent"
          + ?"defprim-tin"


FailToBool(s) = s; !Bool("#t") <+ !Bool("#f")

GetPartRes :
  App(P(x),args) ->
  <try(GetPartRes)>pebody
    where <GetRes>App(P(x),args) => pebody

GetPartRes :
  Info(a,b) ->
  <try(GetPartRes)>b

GetPartRes :
  Let(bindings,Body([],[body])) ->
  <try(GetPartRes)>body

GetPartRes :
  App(Lambda(bindings,Body([],body)),args) ->
  <try(GetPartRes)>body

GetPartRes :
  App(Ofa("cons"),[x,y]) -> App(Ofa("cons"),[x,y])

GetPartRes :
  App(Ova("append"),x) ->
  Quote(Lis1(<map(try(?Quote(Lis1(<id>))))>x))

GetPartRes :
  App(Ova("list"),x) -> Quote(Lis1(x))

GetPartRes :
  App(C(x),y) -> App(C(x),y)

GetRes :
  App(S(x),[arg]) ->
  <index>(<LookupS>S(x),<try(GetPartRes);?App(C(_),<id>)>arg)

GetRes :
  App(ConsQ(x),[App(C(y),args)]) ->
  <FailToBool(<eq>(x,y))>0

GetRes :
  App(ConsQ(x),[arg]) ->
  Bool("#f")
    where not(<?App(C(_),_)>arg)

GetRes :
  app@App(P(x),args) ->
  <FillProc' <+ GetProcRes>app

FillProc' :
  App(P(x),args) ->
  <FillProc>App(P(x),<map(MkStandard)>args)


GetProcRes :
  App(P(x),args) ->
  pebody
    where <LookupFunction>(x,<length>args) => (namedargs,body)
        ; <pe>Define(x,(namedargs,args),body) => pebody

GetRes :
  App(Ofa("abs"),[Num(x)]) ->
  Num(<mul>(-1,x))
    where <lt>(x,0)

GetRes :
  App(Ofa("abs"),[Num(x)]) ->
  Num(x)
    where <gt>(x,0)

GetRes :
  App(Ofa("boolean?"),[cons#(x)]) ->
  <FailToBool(<eq>(cons,"Bool"))>0

GetRes :
  App(Ofa("car"),[arg]) ->
  <?Quote(Lis1(<Hd>))>arg

GetRes :
  App(Ofa("car"),[arg]) ->
  <?App(Ofa("cons"),[<id>,_])>arg

GetRes :
  App(Ofa("car"),[arg]) ->
  <GetRes>App(Ofa("car"),[<?App(Ova("append"),[<id>|_])>arg])

GetRes :
  App(Ofa("car"),[arg]) ->
  <GetRes>App(Ofa("car"),[partres])
    where not(< ?App(Ofa("cons"),[_,_])
              + ?App(Ova("append"),_)
              + ?Quote(Lis1(_))>arg)
; <try(GetPartRes);topdown(try(RmAnnos))>arg => partres

GetRes :
  App(Ofa("cdr"),[arg]) ->
  <?Quote(Lis1(<Tl>));(\ x -> Quote(Lis1(x)) \ )>arg

GetRes :
  App(Ofa("cdr"),[arg]) ->
  <?App(Ofa("cons"),[_,<id>])>arg

GetRes :
  App(Ofa("cdr"),[arg]) ->
  <GetRes>App(Ofa("cdr"),partres)
    where not(<?App(Ofa("cons"),[_,_]) + ?Quote(Lis1(_))>arg)
        ; <GetPartRes;topdown(try(RmAnnos))>arg => partres

GetRes :
  App(Ofa("ceiling"),[Num(num)]) ->
  Num(<int-to-string;string-to-int;inc>num)
    where <gt>(num,0)

GetRes :
  App(Ofa("ceiling"),[Num(num)]) ->
  Num(<int-to-string;string-to-int>num)
    where <leq>(num,0)

GetRes :
  App(Ofa("char->integer"),[Char(c)]) ->
  Num(<explode-string;Hd>c)

GetRes :
  App(Ofa("char-alphabetic?"),[Char(c)]) ->
  <FailToBool(<explode-string;Hd;is-alpha>c)>0

GetRes :
  App(Ofa("char-ci<=?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-ci-leq>(c1,c2))>0

GetRes :
  App(Ofa("char-ci<?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-ci-lt>(c1,c2))>0

GetRes :
  App(Ofa("char-ci=?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-ci-eq>(c1,c2))>0

GetRes :
  App(Ofa("char-ci>=?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-ci-geq>(c1,c2))>0

GetRes :
  App(Ofa("char-ci>?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-ci-gt>(c1,c2))>0

GetRes :
  App(Ofa("char-downcase"),[Char(c)]) ->
  Char(<ToLower>c)
```

```
GetRes :
  App(Ofa("char-lower-case?"),[Char(c)]) ->
  <FailToBool(<explode-string;Hd;is-lower>c)>0

GetRes :
  App(Ofa("char-numeric?"),[Char(c)]) ->
  <FailToBool(<explode-string;Hd;is-num>c)>0

GetRes :
  App(Ofa("char-upcase"),[Char(c)]) ->
  Char(<ToUpper>c)

GetRes :
  App(Ofa("char-upper-case?"),[Char(c)]) ->
  <FailToBool(<explode-string;Hd;is-upper>c)>0

GetRes :
  App(Ofa("char-whitespace?"),[Char(c)]) ->
  <FailToBool(( <eq>(c2,32)
              + <eq>(c2,9)
              + <eq>(c2,10)
              + <eq>(c2,12)
              + <eq>(c2,13)))>0
    where <explode-string;Hd>c => c2

GetRes :
  App(Ofa("char>?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-cd-gt>(c1,c2))>0

GetRes :
  App(Ofa("char>=?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-cd-geq>(c1,c2))>0

GetRes :
  App(Ofa("char=?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-cd-eq>(c1,c2))>0

GetRes :
  App(Ofa("char<?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-cd-lt>(c1,c2))>0

GetRes :
  App(Ofa("char<=?"),[Char(c1),Char(c2)]) ->
  <FailToBool(<string-cd-geq>(c1,c2))>0

GetRes :
  App(Ofa("char?"),[app#(c1)]) ->
  <FailToBool(<eq>(app,"Char"))>0

GetRes :
  App(Ofa("complex?"),[x]) ->
  <FailToBool(<?(Num(_))>x)>0

GetRes :
  App(Ofa("cons"),[Quote(a1),Quote(Lis1(x))]) ->
  Quote(Lis1([a1|x]))

GetRes :
  App(Ofa("cons"),[Quote(a1),Quote(constr#(a2))]) ->
  Quote(Lis2(a1,constr#(a2)))
    where not(<eq>(constr,"Lis1"))

GetRes :
  App(Ofa("cons"),[arg1,Quote(Lis1(x))]) ->
  Quote(Lis1([arg1|x]))
    where <Constant>arg1

GetRes :
  App(Ofa("eof-object?"),[Char(c)]) ->
  <FailToBool(<eq>(<explode-string;Hd>c,3))>0

GetRes :
  App(Ofa("equal?"),[a,b]) ->
  <FailToBool(<eq>(a,b))>0

GetRes :
  App(Ofa("even?"),[Num(x)]) ->
  <FailToBool(<eq>(<mod>(x,2),0))>0

GetRes :
  App(Ofa("exact?"),[Num(num)]) ->
  Bool("#t")

GetRes :
  App(Ofa("floor"),[Num(num)]) ->
  Num(<int-to-string;string-to-int>num)
    where <geq>(num,0)

GetRes :
  App(Ofa("floor"),[Num(num)]) ->
  Num(<int-to-string;string-to-int;dec>num)
    where <lt>(num,0)

GetRes :
  App(Ofa("floor"),[Num(num)]) ->
  Num(<split-before;Fst>(num,"."))

GetRes :
  App(Ofa("gcd"),[Num(num1),Num(num2)]) ->
  Num(<Gcd>(num1,num2))
    where <is-int>num1
; <is-int>num2

Gcd : (0,0) -> 0

Gcd : (i1,i2) -> <Gcd1>(i2,i1,i1)
         where <lt>(i1,i2)
```

```
Gcd : (i1,i2) -> <Gcd1>(i1,i2,i2)
         where <geq>(i1,i2)

Gcd1 : (i1,i2,i3) -> i3
         where <mod>(i1,i3) => 0
             ; <mod>(i2,i3) => 0

Gcd1 : (i1,i2,i3) -> <Gcd1>(i1,i2,<dec>i3)
 where not(<eq>(<mod>(i1,i3),0))
             + not(<eq>(<mod>(i2,i3),0))

GetRes :
  App(Ofa("inexact?"),[Num(num)]) ->
  Bool("#f")


GetRes :
  App(Ofa("integer->char"),[Num(n)]) ->
  Char(<( \ n -> [n] \ );implode-string>n)

GetRes :
  App(Ofa("integer?"),[num]) ->
  <FailToBool(<?Num(<is-int>)>num)>0

GetRes :
  App(Ofa("lcm"),[Num(num1),Num(num2)]) ->
  Num(<Lcm>(num1,num2))
    where <is-int>num1
        ; <is-int>num2

Lcm : (i1,i2) -> <Lcm1>(i2,i1,i1)
         where <lt>(i1,i2)

Lcm : (i1,i2) -> <Lcm1>(i1,i2,i2)
         where <geq>(i1,i2)

Lcm1 : (i1,0,0) -> 0

Lcm1 : (i1,i2,i3) -> i3
          where <mod>(i3,i1) => 0
              ; <mod>(i3,i2) => 0

Lcm1 : (i1,i2,i3) -> <Lcm1>(i1,i2,<inc>i3)
  where not(<eq>(<mod>(i3,i1),0))
              + not(<eq>(<mod>(i3,i2),0))


GetRes :
  App(Ofa("length"),[list]) ->
  res
    where <GetPartRes>list => App(Ofa("cons"),[_,y])
        ; !Num(< (\ x -> App(Ofa("length"),[x]) \ )
               ; GetRes
               ; ?Num(<inc>)>y) => res
```

```
GetRes :
  App(Ofa("length"),[list]) ->
  Num(<try(RmAnnos);?Quote(Lis1(<id>));length>list)

GetRes :
  App(Ofa("list->string"),[list]) ->
  Str(<?Quote(Lis1(<id>));map(?Char(<id>));concat-strings>list)

GetRes :
  App(Ofa("list->vector"),[list]) ->
  Quote(Vec(<?Quote(Lis1(<id>))>list))

GetRes :
  App(Ofa("list?"),[Quote(Lis1(x))]) ->
  Bool("#t")

GetRes :
  App(Ofa("list?"),[Quote(app#(x))]) ->
  Bool("#f")
    where not(<eq>(app,"Lis1"))

GetRes :
  App(Ofa("list?"),[app#(x)]) ->
  Bool("#f")
    where not(<eq>(app,"Quote"))
        ; not(<eq>(app,"App"))

GetRes:
  App(Ofa("list?"),[list]) -> Bool("#t")
    where <GetPartRes>list => App(Ofa("cons"),_)

GetRes :
  App(Ofa("list?"),[list]) -> Bool("#t")
    where <GetPartRes>list => App(Ova("append"),_)

GetRes :
  App(Ofa("modulo"),[Num(num1),Num(num2)]) ->
  Num(<mod>(num1,num2))

GetRes :
  App(Ofa("negative?"),[Num(num)]) ->
  <FailToBool(<lt>(num,0))>0

GetRes :
  App(Ofa("not"),[arg]) ->
  <FailToBool(<eq>(arg,Bool("#f")))>0

GetRes :
  App(Ofa("null?"),[arg]) ->
  <FailToBool(<eq>(arg,Quote(Lis1([]))))>0

GetRes :
  App(Ofa("number?"),[x]) ->
  <FailToBool(<?Num(_)>x)>0

GetRes :
  App(Ofa("odd?"),[Num(x)]) ->
  <FailToBool(<eq>(<mod>(x,2),1))>0

GetRes :
  App(Ofa("pair?"),[Quote(Lis1(a))]) ->
  <FailToBool(not(<eq>(a,[])))>0

GetRes :
  App(Ofa("pair?"),[Quote(Lis2(_,_))]) ->
  Bool("#t")

GetRes :
  App(Ofa("pair?"),[Quote(app#(args))]) ->
  Bool("#f")
    where not(<eq>(app,"Lis1"))
        ; not(<eq>(app,"Lis2"))

GetRes :
  App(Ofa("pair?"),[app#(args)]) ->
  Bool("#f")
    where not(<eq>(app,"Quote"))

GetRes :
 App(Ofa("pair?"),[arg]) ->
 Bool("#t")
    where <GetPartRes>arg => App(Ofa("cons"),_)

GetRes :
  App(Ofa("pair?"),[arg]) ->
  Bool("#t")
    where <GetPartRes>arg => App(Ova("append"),_)

GetRes :
  App(Ofa("positive?"),[Num(num1)]) ->
  <FailToBool(<gt>(num1,0))>0

GetRes :
  App(Ofa("procedure?"),args) ->
  App(Ofa("procedure?"),args)
    where <printnl>(stderr,["Procedure? not defined: ",args])

GetRes :
  App(Ofa("quotient"),[Num(num1),Num(num2)]) ->
  Num(<div>(num1,num2))

GetRes :
  App(Ofa("rational?"),[x]) ->
  <FailToBool(<?Num(<not(is-int)>)>x)>0

GetRes :
  App(Ofa("real?"),[x]) ->
  <FailToBool(<?Num(<not(is-int)>)>x)>0

GetRes :
  App(Ofa("remainder"),[Num(num1),Num(num2)]) ->
  Num(<Remainder>(num1,num2))

Remainder : (i1,i2) -> <subt>(i1,<mul>(<div>(i1,i2),i2))

GetRes :
  App(Ofa("reverse"),[Quote(Lis1(x))]) ->
  Quote(Lis1(<reverse>x))


GetRes :
  App(Ofa("round"),[Num(num)]) ->
  Num(<int-to-string;string-to-int;inc>num)
    where <geq>(num,0)
        ; <geq>(<subt>(num,<int-to-string;string-to-int>num),0.5)

GetRes :
  App(Ofa("round"),[Num(num)]) ->
  Num(<int-to-string;string-to-int>num)
    where <geq>(num,0)
        ; <lt>(<subt>(num,<int-to-string;string-to-int>num),0.5)

GetRes :
  App(Ofa("round"),[Num(num)]) ->
  Num(<int-to-string;string-to-int>num)
    where <lt>(num,0)
        ; <geq>(<subt>(num,<int-to-string;string-to-int>num),-0.5)

GetRes :
  App(Ofa("round"),[Num(num)]) ->
  Num(<int-to-string;string-to-int;dec>num)
    where <lt>(num,0)
        ; <lt>(<subt>(num,<int-to-string;string-to-int>num),-0.5)

GetRes :
  App(Ofa("string->list"),[Str(str)]) ->
  < explode-string
  ; map(\x -> Char(<implode-string>[x]) \ )
  ; (\x -> Quote(Lis1(x)) \ )>str

GetRes :
  App(Ofa("string->symbol"),[Str(str)]) ->
  Sym(str)

GetRes :
  App(Ofa("string-ci<=?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-ci-leq>(str1,str2))>0

GetRes :
  App(Ofa("string-ci<?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-ci-lt>(str1,str2))>0
```

```
GetRes :
  App(Ofa("string-ci=?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-ci-eq>(str1,str2))>0

GetRes :
  App(Ofa("string-ci>=?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-ci-geq>(str1,str2))>0

GetRes :
  App(Ofa("string-ci>?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-ci-gt>(str1,str2))>0

GetRes :
  App(Ofa("string-length"),[Str(str)]) ->
  Num(<string-length>str)

GetRes :
  App(Ofa("string-ref"),[Str(str),Num(num)]) ->
  Char(<index;( \ x -> [x] \ );implode-string>
    (<inc>num,<explode-string>str))

GetRes :
  App(Ofa("string<=?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-cd-leq>(str1,str2))>0

GetRes :
  App(Ofa("string<?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-cd-lt>(str1,str2))>0

GetRes :
  App(Ofa("string=?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-cd-eq>(str1,str2))>0

GetRes :
  App(Ofa("string>?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-cd-gt>(str1,str2))>0

GetRes :
  App(Ofa("string>=?"),[Str(str1),Str(str2)]) ->
  <FailToBool(<string-cd-geq>(str1,str2))>0

GetRes :
  App(Ofa("string?"),[constr#(args)]) ->
  <FailToBool(<eq>(constr,"Str"))>0

GetRes :
  App(Ofa("substring"),[Str(str),Num(num1),Num(num2)]) ->
  Str(<Substring>(str,num1,num2))

Substring :
  (str,begin,end) ->
  <Substring1;implode-string>
    (<explode-string>str,[],<inc>begin,<inc>end)
```

```
Substring1 :
  (list,res,begin,end) ->
  <Substring1>(list,[<index>(end,list)|res],begin,<dec>end)
    where <lt>(begin,end)

Substring1 :
  (list,res,begin,end) ->
  [<index>(end,list)|res]
    where <eq>(begin,end)

GetRes :
  App(Ofa("symbol->string"),[Sym(sym)]) ->
  Str(sym)

GetRes :
  App(Ofa("symbol?"),[constr#(args)]) ->
  <FailToBool(<eq>(constr,"Sym"))>0

GetRes :
  App(Ofa("truncate"),[Num(num)]) ->
  Num(<int-to-string;string-to-int>num)

GetRes :
  App(Ofa("vector->list"),[vec]) ->
  Quote(Lis1(<?Quote(Vec(<id>))>vec))

GetRes :
  App(Ofa("vector-length"),[Quote(Vec(lis))]) ->
  Num(<length>lis)

GetRes :
  App(Ofa("vector?"),[Quote(constr#(args))]) ->
  <FailToBool(<eq>(constr,"Vec"))>0

GetRes :
  App(Ofa("vector?"),[constr#(args)]) ->
  Bool("#f")
    where not(<eq>(constr,"Quote"))

GetRes :
  App(Ofa("zero?"),[Num(num)]) ->
  <FailToBool(<eq>(num,"0"))>0

string-ci-gt = string-gt

string-ci-lt = string-lt

string-ci-eq :
  (c1,c2) -> (c1,c2)
    where <eq>(<ToUpper>c1,<ToUpper>c2)

string-ci-geq :
 (c1,c2) -> (c1,c2)
```

```
    where ( <string-ci-gt>(c1,c2)
         <+ <string-ci-eq>(c1,c2))

string-ci-leq :
 (c1,c2) -> (c1,c2)
    where ( <string-ci-lt>(c1,c2)
         <+ <string-ci-eq>(c1,c2))

string-cd-gt =
  try((explode-string, explode-string));
  strcmp; ?1

string-cd-lt =
  try((explode-string, explode-string));
  strcmp; ?-1

string-cd-eq = ?(c1,c1)

string-cd-geq :
 (c1,c2) -> (c1,c2)
    where (<string-cd-gt>(c1,c2) + <string-cd-eq>(c1,c2))

string-cd-leq :
 (c1,c2) -> (c1,c2)
    where (<string-cd-lt>(c1,c2) + <string-cd-eq>(c1,c2))

ToUpper : s -> <explode-string;map(to-upper);implode-string>s
ToLower : s -> <explode-string;map(to-lower);implode-string>s


GetRes :
  App(Ova("+"),y) ->
  Num(res)
    where <foldr(!0, add, ?Num(<id>))> y => res

GetRes :
  App(Ova("*"),y) ->
  Num(res)
    where <foldr( !1, mul, ?Num(<id>))> y => res

GetRes :
  App(Ova("/"),y) ->
  Num(res)
    where <foldr( !1, div, ?Num(<id>))> y => res

GetRes :
  App(Ova("-"),y) ->
  Num(res)
    where <foldr( !0, subt, ?Num(<id>))> y => res

GetRes :
  App(Ova("<"),y) ->
  <FailToBool(<map(?Num(<id>));foldr1(lt;Fst)>y)>0
```

```
GetRes :
  App(Ova("<="),y) ->
  <FailToBool(<map(?Num(<id>));foldr1(leq;Fst)>y)>0

GetRes :
  App(Ova("="),y) ->
  <FailToBool(<foldr1(eq;Fst)>y)>0

GetRes :
  App(Ova(">="),y) ->
  <FailToBool(<map(?Num(<id>));foldr1(geq;Fst)>y)>0

GetRes :
  App(Ova(">"),y) ->
  <FailToBool(<map(?Num(<id>));foldr1(gt;Fst)>y)>0

GetRes :
  App(Ova("append"),y) ->
  Quote(Lis1(<map(?Quote(Lis1(<id>)));concat>y))

GetRes :
  App(Ova("list"),y) ->
  Quote(Lis1(<map(try(?Quote(<id>)))>y))

GetRes :
  App(Ova("make-string"),[Num(num)]) ->
  Str(new-string)
    where <MakeString>(" ","",num) => new-string

GetRes :
  App(Ova("make-string"),[Num(num),Char(char)]) ->
  Str(new-string)
    where <MakeString>(char,"",num) => new-string

MakeString : (a,b,0) -> b

MakeString :
  (a,b,int) ->
  <MakeString>(a,<conc-strings>(a,b),<dec>int)
    where not(<eq>(int,0))

GetRes :
  App(Ova("make-vector"),[Num(num),x]) ->
  Quote(Vec(new-vector))
    where <MakeVector>(x,[],num) => new-vector

GetRes :
  App(Ova("make-vector"),[Num(num)]) ->
  Quote(Vec(new-vector))
    where <MakeVector>(V("dummy"),[],num) => new-vector

MakeVector : (a,b,0) -> b
MakeVector :
  (a,b,int) ->
  <MakeVector>(a,<conc>([a],b),<dec>int)
    where not(<eq>(int,0))

GetRes :
  App(Ova("max"),[Quote(Lis1(y))]) ->
  Num(<map(?Num(<id>));list-max>y)

GetRes :
  App(Ova("min"),[Quote(Lis1(y))]) ->
  Num(<map(?Num(<id>));list-min>y)

GetRes :
  App(Ova("number->string"),[Num(num)]) ->
  Str(<num-to-string>num)

GetRes :
  App(Ova("string"),y) ->
  Str(<map(?Char(<id>));concat-strings>y)

GetRes :
  App(Ova("string->number"),[Str(str)]) ->
  Num(<string-to-num>str)

GetRes :
  App(Ova("string-append"),y) ->
  Str(<map(?Str(<id>));concat-strings>y)

GetRes :
  App(Ova("vector"),y) ->
   Quote(Vec(y))

string-to-num = string-to-int <+ string-to-real
num-to-string = real-to-string <+ int-to-string
```

Appendix F

VarRenamer.r

```
module VarRenamer
imports Similix TemporaryAnnotations lib dynamic-rules

strategies
  renamevars =
    rec r( try(Let(map((id,r)),id))
         ; try(LetRec(map(MkRecLet),id))
         ; {| RenameVar,
              RenameStr :
              try( RenameVars
                 + RenameVar
                 + RenameStr
                 )
             ; (Let(map((r,id)),r) <+ all(r))
            |})
rules

  RenameVars =
    ?Let(bindings,body)
  ; <map(RenameBindingVars)>bindings

  RenameBindingVars =
    ?(x,y)
  ; <RenameVar>V(x)
  ; new => z
  ; <CreateRules>(x,z)

  RenameBindingVars =
    ?(x,y)
  ; not(<RenameVar>V(x))
  ; <CreateRules>(x,x)

  RenameVars =
    ?Define(name,args,body)
  ; <zip(CreateRules)>(args,args)

  RenameVars =
    ?Lambda(names,body)
  ; <map(try(VarRenamer))>names

  MkRecLet :
```

```
  (a,b,c) -> RecLet(a,b,c)

RenameVars :
 RecLet(procname,args,body) ->
 (procname,args,body)
   where <map(try(VarRenamer))>args

VarRenamer =
  ?x
; <RenameVar>V(x)
; new => z
; <CreateRules>(x,z)

VarRenamer =
  ?x
; not(<RenameVar>V(x))
; <CreateRules>(x,x)

CreateRules =
  ?(a,b)
; rules(RenameVar : V(a) -> V(b)
        RenameStr : a -> b)
```

Appendix G
Preparator.r

```
module Preparator
imports Similix lib

strategies

  prepare =
    topdown(try(MakeFunctionsLookupable));
    topdown(try(ConstrSelRules))

rules

  MakeFunctionsLookupable =
    ?Define(name,args,body)
    ; <length>args => number
    ; rules( LookupFunction:
               (name,number) ->
               (args,body) )

  ConstrSelRules =
    ?Loadt(filename)
    ; <ParseSimilix>filename => contents
    ; < ?OCDs(<id>)
      ; filter(Defconstr(id))
      ; map(?Defconstr(<id>))
      ; concat>contents => temp
    ; map(MkConstrSelNames) => constrselnames
    ; <map(Snd)>temp => temp2
    ; < zip(zip(Ss2OrC))
      ; MkConstrSelRules>(temp2,constrselnames)

    // Parse a similix file and produce its abstract syntax tree

  ParseSimilix :
    filename -> contents
    where new-file => newfile1
        ; new-file => newfile2
        ; <parse-similix>(<un-double-quote>filename, newfile1)
        ; <call>("implode-asfix", ["-i", newfile1, "-o", newfile2])
        ; <ReadFromFile> newfile2 => contents
        ; <call>("rm",["-f",newfile1,newfile2])
```

```
// Call sglr to parse a similix file
// Note that the .tbl is expected in the current directory

parse-similix =
  ?(file1, file2);
  <call>("sglr", ["-2","-p","Similix.tbl","-i",file1,"-o",file2])

MkConstrSelRules =
  ?a
  ; < zip(zip(id))
    ; concat
    ; map(MkConstrSelRule)>(a,<map(length;dec;upto;map(inc))>a)

MkConstrSelRule =
  ?(a,b)
  ; rules(LookupS : S(a) -> b)

Ss2OrC :
  ("*",b) -> b

Ss2OrC :
  (Ss2(a),b) -> a

MkConstrSelNames :
  (a,b) -> constrselnames
    where <length;dec;upto;map(int-to-string)>b => nums
        ; <zip(ConcatNames)>
            (<map( \ x -> a \ )>nums,nums)
               => constrselnames

ConcatNames :
  (a,b) -> <concat-strings>[a,".",b]
```

Appendix H
NameCreator.r

```
module NameCreator
imports Similix lib

rules

  BtaName :
    App(P(x), args) ->
    Name(x, <concat-strings>["-a" | <map(MkBTAName)> args])

  MkBTAName :
    exp{annos} ->
    <conc-strings>("-",<MkAnnotatedName>annos)

  MkAnnotatedName :
    annos ->
    <concat-strings>[ "<"
                    , <map(MkAnnotatedName); concat-strings>annos
                    , ">"]
      where <is-list>annos

  MkAnnotatedName :
    LisAnno(annos) ->
    <concat-strings>[ "<["
                    , <map(MkAnnotatedName); concat-strings>annos
                    , "]>"]

  MkAnnotatedName :
    Annotation(annos) ->
    <MkAnnotatedName>annos

  MkAnnotatedName :
    St -> "s"

  MkAnnotatedName :
    Dy -> "d"

  PeName :
    App(P(Name(x,_)),args) ->
    Name(x,<concat-strings>["-pe" | <map(MkPEName <+ !"-_")> args])

  MkPEName :
    constr#([arg]) -> <conc-strings>("-",arg)
      where <Constant>constr#([arg])

  MkPEName :
    V(a) -> "-_"

  MkPEName :
    App(arg#(x),y) -> "-_"
      where not(<eq>(arg,"C"))

  MkPEName :
    Lis1(x) ->
    <concat-strings>[ "-<["
                    , <map(MkPEName);concat-strings>x
                    , "]>"]

  MkPEName :
    Lis2(x,y) ->
    <concat-strings>[ "-<["
                    , <map(MkPEName);concat-strings>x
                    , <MkPEName>y
                    , "]>"]

  MkPEName :
    App(C(x),args) ->
    <concat-strings>[ "-<"
                    , <map(MkPEName);concat-strings>args
                    , ">"]

  MkPEName :
    Quote(x) -> <MkPEName>x

  MkPEName :
    Info(x,y) -> "-_"

  MkPEName :
    x -> "-_"
where <is-string>x
```

Appendix I

Annotator.r

```
module Annotator
imports Similix lib

rules

  Annotate =
    MkConstantsStatic
 <+ MkConstantsStatic2
 <+ MkVarsDynamic

  MkConstantsStatic :
    a{b} -> a{b}

  MkConstantsStatic :
    x ->
    x{St}
      where <Constant>x

  MkConstantsStatic :
    Lis2(list,pt2) ->
    Lis2(annlist,annpt2){LisAnno(annos)}
      where <map(Annotate)>list => annlist
          ; <Annotate>pt2 => annpt2
          ; <conc>(<map(RmVals)>annlist,[<RmVals>annpt2]) => annos

  MkConstantsStatic :
    Lis1(list) ->
    Lis1(annlist){LisAnno(annos)}
      where <map(Annotate)>list => annlist
          ; <map(RmVals)>annlist => annos

  MkConstantsStatic :
    lis@Quote(Lis1(list)) ->
    Quote(Lis1(annlist)){annos}
      where <map(Annotate)>list => annlist
          ; <map(RmVals)>annlist => annos

  MkConstantsStatic :
    Quote(x) -> Quote(annx){anns}
      where <Annotate>x => annx
          ; <GetAnnotation>annx => anns
```

```
MkConstantsStatic2 :
  App(C(x),args) ->
  App(C(x),annargs){[St | annos]}
    where <map(Annotate)>args => annargs
        ; <map(RmVals)>annargs => annos

MkVarsDynamic :
  V(x) ->
  V(x){Dy}

MkVarsDynamic :
  x -> x{Dy} where <is-string>x

Constant =
  ?Str(_)
+ ?Char(_)
+ ?Num(_)
+ ?Bool(_)

GetAnnotation = get-annos;?[<id>]

RmAnnos = ?x{_};!x{}
RmVals  = ?a{b};!b
```