URL: http://www.elsevier.nl/locate/entcs/volume65.html 20 pages

# Building Interpreters with Rewriting Strategies

Eelco Dolstra and Eelco Visser

*Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089, 3508 TB Utrecht, The Netherlands. eelco@cs.uu.nl, visser@acm.org, http://www.cs.uu.nl/people/{eelco,visser}/*

**Abstract**

Programming language semantics based on pure rewrite rules suffers from the gap between the rewriting strategy implemented in rewriting engines and the intended evaluation strategy. This paper shows how programmable rewriting strategies can be used to implement interpreters for programming languages based on rewrite rules. The advantage of this approach is that reduction rules are first class entities that can be reused in different strategies, even in other kinds of program transformations such as optimizers. The approach is illustrated with several interpreters for the lambda calculus based on implicit and explicit (parallel) substitution, different strategies including normalization, eager evaluation, lazy evaluation, and lazy evaluation with updates. An extension with pattern matching and choice shows that such interpreters can easily be extended.

## 1 Introduction

Language prototyping is the activity of designing and validating the design of programming languages. Prototype interpreters are useful for experimentation with language design, to provide a reference implementation for validation of compilers, and as basis for the design and implementation of optimizers and other program transformation systems. A language prototyping methodology should support executability of language definitions and easy modification of a definition to include or exclude language constructs and experiment with different evaluation strategies.

Systems supporting language prototyping can be divided into systems based on a denotational approach and systems based on an operational approach. In the denotational approach a program expression is translated to its computational value. Typical systems supporting this approach are action semantics, LPS, and Montages. In action semantics [5] an expression is translated to an action, which represents its computation. In the LPS system [11] an expression is mapped onto a monadic computation [12]. In Montages programs are translated to abstract state machines [10]. While denotational semantics is very useful for building compilers, it does not provide a theory of syntactic manipulation of programs at the source level, which is needed for building source-to-source transformation systems.

In the operational approach a program expression is directly interpreted by some kind of transition system to produce its value. This provides a direct operational interpretation of the (abstract) syntactic constructs of a language that can be used to build interpreters, but can also be used as the basis for implementing other program transformations, such as optimizers, refactoring engines, and software renovation tools.

There are different styles of operational semantics. In natural semantics [9] supported by the CENTAUR system [3] an expression is directly related to its value, possibly through the use of recursive invocations of the evaluation relation. In rewriting semantics, supported by many systems including ASF+SDF [4] and ELAN [2], an expression is normalized by successive applications of reduction rules, which define a, generally small, modification of an expression. In the approach of syntactic theories [7], supported for example by the SL language [17,18], a set of reduction rules is extended with evaluation contexts that limit the positions in which reduction rules can be applied. Thus, evaluation contexts overcome a limitation of pure rewriting based semantics. Exhaustive application of reduction rules is usually not an adequate description of the semantics of a language. Therefore, the gap between evaluation strategy and rewriting strategy (provided by the rewriting engine) is usually filled by explicitly specifying an evaluation function instead of using pure rewrite rules.

In this paper we show how the paradigm of programmable rewriting strategies can be used to implement interpreters. In this approach the semantics is defined by means of 'constant folding' reduction rules, which describe valid transformations on program expressions. In addition, a user-defined rewriting strategy determines in what order and at which positions in an expression the rules are applied, thus determining the evaluation strategy.

This approach is attractive since semantic rules are first-class entities which are reusable and can be combined using different strategies, making it easy to experiment with evaluation strategy and language configurations. Such configurations do not depend on the module structure of the specification—as is the case, for example, in the approach to modular action semantics [5]; it is possible to define *different* interpreters in the same specification module based on the same set of rules. For example, in [6] interpreters based on lazy and strict evaluation for RhoStratego are defined using the same set of rules. Furthermore, the set of rules can be used for other program transformations than complete evaluation of a program by using a different strategy. Indeed, the approach stems from the implementation of *optimizers* with rewriting strategies [16].

We illustrate the approach by defining a number of interpreters for lambda expressions. Although the lambda calculus is not a full fledged programming language, it is useful to illustrate various solutions to the key problems in defining symbolic interpreters, i.e., the treatment of variable bindings and substitutions and the reduction order.

In Section 2 we define lambda reduction with a substitution meta-operation and with substitution defined explicitly through reduction rules. In Section 3 we consider the specification of evaluation strategies based on reduction rules with explicit substitution. Strategies include exhaustive application, eager and lazy evaluation, parallel substitution, and lazy evaluation with updates. In Section 4 explicit substitution is replaced with the generation of dynamic rewrite rules. In Sections 5 and 6 we extend the basic lambda language with pattern matching, pattern match failure, and choice, leading to an interpreter for the core of the RhoStratego language [6]. Section 7 discusses applications of this approach.

All examples are specified in Stratego, a language for program transformation based on the paradigm of rewriting strategies [14]. We explain the Stratego constructs used, but not in depth. For a full account see [14,13,16].

## 2 Lambda Reduction with Explicit Substitution

In this section we explore various ways of defining interpreters for lambda expressions using explicit substitution.

Lambda expressions are composed of variables, applications of lambda expressions, and abstractions of variables from lambda expressions. To describe the structure of expressions, we will use in this paper both concrete syntax following context-free grammar productions and abstract syntax following constructor declarations. Thus, lambda expressions are described by the following algebraic signature, with concrete syntax in comments:

```
signature
  sorts Exp
  constructors
    Var : String -> Exp          // Id -> Exp
    App : Exp * Exp -> Exp       // Exp Exp -> Exp
    Abs : String * Exp -> Exp    // "\\" Id "->" Exp -> Exp
```

Stratego specifications define transformations on abstract syntax terms. However, concrete syntax can be regarded as syntactic sugar for abstract syntax expressions [15]. Concrete syntax terms will be written between `[[` and `]]`. Thus, `[[ (\x -> x) y ]]` denotes the term `App(Abs("x", Var("x")), Var("y"))`. We will use concrete syntax in rules, but abstract syntax in the definition of traversals.

### *2.1 Beta Reduction*

The meaning of lambda expressions can be expressed by reduction rules that reduce an expression to a canonical form. The canonical form represents the *value* of the expression. The reduction of lambda expressions is expressed by the single *beta reduction* rule

```
  Beta:
    [[ (\ x -> e1) e2 ]] ->
                        <substitute> ([[ x ]], [[ e2 ]], [[ e1 ]])
```

where the right-hand side denotes the (capture-free) substitution of $x$ by $e_2$ in $e_1$. A lambda expression without $\beta$ redices is in normal form and represents the value of all expressions that can be reduced to it. Of course, this normal form should be considered modulo renaming of bound variables.

A pure rewriting-based interpreter for lambda expressions exhaustively applies rule `Beta` to a lambda expression in arbitrary order. This can be expressed by the `reduce` strategy, which is defined as

```
  reduce(s) =
    repeat(rec x(some(x) + s))
```

The `reduce` strategy takes a selection of rules (`s`) and repeatedly tries to find some subterms for which `s` succeeds. Using this strategy a lambda interpreter can be defined as follows:

```
  eval1 = reduce(Beta)
```

In the next section we will consider strategies that are more in line with interpreters for actual programming languages.

### *2.2 Substitution Laws*

In the above formulation of beta reduction, capture-free substitution has not been defined as part of the reduction, but is assumed as a meta-operation. Besides the fact that this does not provide a satisfactory definition of reduction, this has consequences for the complexity of the interpretation algorithm. The implementation of the substitution operation must traverse the term, while the rewriting strategy itself also traverses the term, resulting in multiple traversals. Therefore, it is desirable to express substitution as part of the reduction process itself.

To make substitution explicit we extend the signature of lambda expressions with a `let` binding to be used as substitution operator. Let bindings have the following syntax:

```
signature
  constructors
    Let : String * Exp * Exp -> Exp
                             // "let" Id "=" Exp "in" Exp -> Exp
```

A let binding `let x = e1 in e2` binds the expression `e1` to the variable `x`. The binding holds in the expression `e2`, not in `e1`.

Again, the meaning of the let construct can be expressed by means of reduction rules. When the `Let` body is just a variable, the let bound expression is substituted for the variable if it is the same as the bound variable

```
SubsVar :
  [[ let x = e in x ]] -> [[ e ]]
```

otherwise, the binding is eliminated.

```
SubsVar :
  [[ let x = e in y ]] -> [[ y ]]
  where <not(eq)> (x, y)
```

Substitution distributes over application

```
SubsApp :
  [[ let x = e in (e1 e2) ]] ->
  [[ (let x = e in e1) (let x = e in e2) ]]
```

Substitution under abstraction needs to be treated with care to avoid capture of free variables in the let-bound expression. If the abstraction variable is the same as the let-bound variable, the abstraction shadows the let binding, which can thus be eliminated.

```
SubsAbs :
  [[ let x = e1 in \ x -> e2 ]] -> [[ \ x -> e2 ]]
```

If the variables are not the same, the substitution can be applied to the body of the abstraction, but after renaming the abstraction variable to a new name.

```
SubsAbs :
  [[ let x = e1 in \ y -> e2 ]] ->
  [[ \ z -> (let x = e1 in let y = z in e2) ]]
  where <not(eq)> (x, y); new => z
```

The strategy `new` generates a new unique name. The name is guaranteed not to coincide with any string currently in use.

Given these reduction rules, we can redefine beta reduction by translating the application of an abstraction to a substitution of the actual parameter for the formal parameter by means of a let binding:

```
BetaES:
  [[ (\ x -> e1) e2 ]] -> [[ let x = e2 in e1 ]]
```

Beta reduction with explicit substitution can now be defined without recourse to meta-operations by reducing with respect to the `BetaES` rules and the `Subs*` rules. By giving a name (`lambda-es`) to this selection of rules it can be reused later on.

```
eval2 =
  reduce(lambda-es)
lambda-es =
  BetaES + SubsVar + SubsApp + SubsAbs
```

## 3 Evaluation Strategies

Now that we have made substitution explicit, we turn to the issue of reduction order. The arbitrary reduction order of the `reduce` strategy is not very efficient, nor does it describe the strategy that should be followed by a programming language interpreter.

First of all we consider the efficiency issue. The `reduce` strategy repeatedly looks for several redices, starting at the root of the term. A more efficient strategy is the following `innermost` strategy that completely normalizes subterms before it considers the term itself:

```
innermost(s) =
  rec x(memo(all(x); try(s; x)))
```

The strategy first normalize the subterms (`all(x)`), and only then tries to apply one of the rules (`s`). If that succeeds, it recursively reduces the result of applying the rule. Otherwise the term is in normal form. To avoid renormalizing terms, the `memo` operator memorizes normalization results. See [8] for an optimization of the innermost strategy that does not depend on memorization.

Reusing the same set of rules we can now define a more efficient evaluation strategy based on `innermost`:

```
eval3 = innermost(lambda-es)
```

### *3.1 Eager Evaluation*

Exhaustive application of the evaluation rules is not adequate for an interpreter of a programming language. For example, reducing the body of a function is often only meaningful after its parameters are known; it could contain non-terminating reductions. To improve the termination behavior of programs, interpreters restrict the evaluation of expressions. Such restrictions can be expressed by adapting the strategy for applying the evaluation rules.

An eager (or strict) interpreter evaluates the let-bound expression before substituting it into the body of the let, evaluates arguments to functions, but does not evaluate bodies of functions. This is basically the same as a complete reduction, except that the traversal should be limited in order not to apply any reductions under abstractions. This is expressed by the following evaluation strategy, which follows the innermost algorithm, but restricts traversal:

```
eval4 = rec e(
  (Var(id) + App(e, e) + Abs(id, id) + Let(id, e, e));
  try(lambda-es; e)
)
```

The traversal is defined using *congruence operators* that specify traversal specific for a constructor. For example, the `App(e,e)` strategy specifies that `e` should be applied to the two direct subterms of `App` terms. On the other hand `Abs(id,id)` indicates that the identity transformation should be applied to the subterms of `Abs` terms, i.e., the subterms of `Abs` should not be evaluated.

A more economic definition of this strategy only specifies the constructors that should be traversed, not touching any of the other constructors:

```
eval5 = rec e(
  try(App(e, e) + Let(id, e, e));
  try(lambda-es; e)
)
```

### *3.2 Lazy Evaluation*

Lazy evaluation further improves the termination behavior by only reducing expressions whose value is needed. In the lambda calculus, values are only ever needed when performing a beta reduction, therefore the arguments of applications and let bound expressions should not be evaluated; only function positions of applications should be evaluated to expose possible redices. This is expressed in the following strategy, which further restricts traversal:

```
eval6 = rec e(
  try(App(e, id) + Let(id, id, e));
  try(lambda-es; e)
)
```

The only difference with the eager evaluation strategy is the restriction of the traversal, the evaluation rules used remain the same.

This strategy implements lazy evaluation or *call by need* since an expression is only evaluated when it is needed. Usually lazy evaluators also memorize the value of an expression after evaluation such that it is only evaluated once. This aspect will be discussed later on.

### *3.3 Parallel Substitution*

The formulation of explicit substitution defined above, distributes a single variable binding at a time over the constructors of an expression. By combining multiple bindings into a set of bindings, the number of term traversals can be reduced. For this purpose we introduce the parallel let (`letp`) construct with a list of bindings instead of a single binding. The syntax of the construct is:

```
signature
  constructors
    Letp : List(Bind) * Exp -> Exp
                       // "letp"  {Bind ";"}* "in" Exp -> Exp
    Bind : String * Exp -> Bind             // Id "=" Exp -> Bind
```

The first argument of a `letp` is a list binding of expressions to variables.

The new `letp` construct requires the reformulation of the evaluation rules. Variable evaluation entails looking up the value in the list of bindings. If no binding is found (`lookupb` fails), the original variable is produced.

```
  PSubsVar :
    [[ letp ds in x ]] -> [[ e ]]
    where (<lookupb>(x, ds) <+ ![[ x ]]) => e

  lookupb =
    ?(x, ds); <fetch(?[[ x = e1 ]]); ![[ e1 ]]> ds
```

Distribution of substitution over application distributes all bindings in one step:

```
  PSubsApp :
    [[ letp ds in e1 e2 ]] ->
    [[ (letp ds in e1) (letp ds in e2) ]]
```

Distribution of substitution over abstraction requires renaming of the binding variable to avoid free variable capture.

```
  PSubsAbs :
    [[ letp ds in \ x -> e ]] ->
    [[ \ y -> (letp ds in letp x = y in e) ]]
    where new => y
```

Beta reduction consists of binding the argument to the formal parameter:

```
  BetaPES :
    [[ (\ x -> e1) e2 ]] ->
    [[ letp y = e2 in letp x = y in e1 ]]
    where new => y
```

To achieve the benefit of parallel substitution, the bindings of two adjacent `letp`s can be combined. This requires applying the substitutions of the outer `letp` to the let-bound expressions of the inner `letp`; this corresponds to the substitution lemma. Rule `LetLetOne` defines the combination of two `letp`s with a single binding:

```
  LetLetOne:
    [[ letp x = e1 in (letp y = e2 in e3) ]] ->
    [[ letp y = (letp x = e1 in e2); x = e1 in e3 ]]
```

Rule `LetLet` generalizes this to `letp`s with arbitrary numbers of bindings:

```
  LetLet:
    [[ letp ds1 in letp ds2 in e1 ]] ->
    [[ letp ds4 in e1 ]]
    where <map(\ [[ x = e ]] ->
                 [[ x = letp ds1 in e ]] \ )> ds2 => ds3
        ; <conc> (ds3, ds1) => ds4
```

Finally we give the collection of reduction rules for parallel explicit substitution a name:

```
lambda-pes = BetaPES + PSubsVar + PSubsApp + PSubsAbs + LetLet
```

Using these rules we can define evaluation strategies along the same lines as before. The eager evaluation strategy evaluates function arguments and `letp` bindings:

```
eval7 =
  rec e(try(App(e, e) + Letp(map(Bind(id,e)), id))
       ; rec e(try(lambda-pes; rec y(try(App(y, y)); e))))
```

The inner (`y`) recursion only traverses application nodes, and not `letp` bindings to avoid re-evaluating substitutions.

The lazy evaluation evaluation strategy only evaluates expressions in function positions:

```
eval8 = rec e(
  try(App(e, id)); try(lambda-pes; e)
)
```

Note that it is no longer necessary to evaluate the body of `letp` expressions since the substitution rules will push the `letp` inside first.

## *3.4 Sharing Bindings*

A problem with the explicit substitution approach above is that substitutions are distributed over the term by rule `(P)SubsApp`. In the case of eager evaluation this is no problem, since the expressions bound in the substitutions are evaluated first. In the lazy evaluation strategies (`eval6`,`eval8`), however, the let-bound expressions are not evaluated. This guarantees that expressions are only evaluated when needed, but the duplication by the substitution distribution may result in multiple evaluations of the same expression. A good lazy evaluator will update the binding to a variable as soon as it has been evaluated once. This requires maintaining the bindings after substitution.

To formalize the notion of updating we use the set of bindings introduced for parallel substitution as a global heap on which variable bindings are collected. At the start of evaluation an empty heap is introduced by rule `LetLift`:

```
LetLift:
  [[ e ]] -> [[ letp in e ]]
```

Variable substitution is the same as before, but should maintain the bindings and should only reduce when the variable is present to avoid non-termination

```
USubsVar :
  [[ letp ds in x ]] -> [[ letp ds in e ]]
  where <lookupb>(x, ds) => e
```

Evaluation of the function position of an application can no longer be expressed using a simple congruence operator since it needs to distribute the substitution environment and restore the environment afterwards:

```
LetAppL(eval) :
  [[ letp ds1 in e1 e2 ]] -> [[ letp ds2 in e3 e2 ]]
  where <eval> [[ letp ds1 in e1 ]] => [[ letp ds2 in e3 ]]
```

The renaming of the variables bound in the inner `Let` in the `LetLetRen` rule corresponds to allocating space on the heap for these bindings.

```
LetLetRen:
  [[ letp ds1 in letp ds2 in e1 ]] ->
  [[ letp ds4 in e2 ]]
  where <rename> [[ letp ds2 in e1 ]] => [[ letp ds3 in e2 ]]
      ; <conc> (ds3, ds1) => ds4
```

With these rules we can define a lazy evaluation strategy that threads the evaluation environment, but does not yet update any bindings. Every time a variable is substituted for, the expression bound to it is reevaluated.

```
eval9 =
  LetLift; rec e(
    try(LetAppL(e));
    try((USubsVar + Letp(id, BetaPES) + LetLetRen); e)
  )
```

This strategy can be extended to update variable bindings using the additional rule `Update`, which takes as parameter an evaluation strategy.

```
Update(eval) :
  [[ letp ds1 in x ]] -> [[ letp x = e; ds2 in e ]]
  where <eval> [[ letp ds1 in x ]] => [[ letp ds2 in e ]]
```

The evaluation strategy `e` is used to evaluate the variable, resulting in a value `v`. The value is produced as result, as well as bound to the variable `x` in the environment. Subsequent evaluations of variable `x` will directly produce the value.

The `Update` rule can be used to adapt strategy `eval9` into an updating strategy:

```
eval10 =
  LetLift; rec e(
    try(LetAppL(e));
    try(Update(USubsVar; e)
        + (Letp(id, BetaPES) + LetLetRen); e)
  )
```

The strategy passed to `Update` first performs the substitution and then recursively evaluates the resulting expression.

Thus we have specified a full operational account of lazy lambda evaluation. The scheme can be extended with operational notions such as garbage collection and blackholing to detect infinite loops [6].

## 4 Lambda Reduction with Dynamic Rules

The explicit substitution interpreters discussed in the previous sections provide a full account of operational notions of lambda calculus evaluation. However, the manipulation of substitutions as part of the term structure requires bookkeeping and plumbing, and distracts from the basic scheme of the evaluation traversal. This is exemplified by the change from a simple congruence operator for expression traversal through an `App` node to the environment threading rule `LetAppL`.

This situation can be improved by noting that a substitution can be considered as a rewrite rule that rewrites a variable to the expression with which it should be substituted. Ordinary rewrite rules cannot be used, however, since the set of bindings is extended dynamically during evaluation. In this section we replace the explicit substitution environment by a set of dynamic rewrite rules. A dynamic rewrite rule [13] is a rewrite rule that is generated at run-time and that can inherit (meta-)variable bindings from its generation context.

### *4.1 Eager Evaluation*

The application of an abstraction gives rise to a substitution. Using dynamic rules this is expressed by rewriting the application to the body of the abstraction and generating a rule `EvalVar` that rewrites the abstraction variable to the argument of the application.

```
EvalApp :
  [[ (\ x -> e1) e2 ]] -> [[ e3 ]]
  where
     <rename> [[ \ x -> e1 ]] => [[ \ y -> e3 ]];
     rules(EvalVar : [[ y ]] -> [[ e2 ]])
```

Since the abstraction is opened up by the transformation rule, the bound variable needs to be renamed in order to prevent clashes with free variables.

The construct `rules(EvalVar : [[ y ]] -> [[ e2 ]])` generates a rule `EvalVar`, which rewrites an occurrence of variable `y` to `e2`, for the terms bound to `y` and `e2` in the context, in this case the left-hand side of the `EvalApp` rule. If prior to this generation, another `EvalVar` rule would have been generated for the same binding to `y`, the new rule would override the old one. To regulate such overrides the dynamic rule mechanism provides a scope construct, which can be used to determine how long a dynamic rule should remain valid. However, in the case of the evaluation strategy we are defining, this is not needed; by renaming the abstraction before generating the dynamic rule, we are guaranteed that `y` is a fresh variable and does not clash with any earlier bindings.

If let bindings are not used to express substitutions, they can be evaluated just like applications of abstractions.

```
EvalLet:
  [[ let x = e1 in e2 ]] -> [[ (\ x -> e2) e1 ]]
```

An eager evaluation strategy based on these rules is similar to an eager evaluation strategy with explicit substitution, i.e., the expression is traversed in a restricted fashion after which the evaluation rules are applied. The dynamic rule `EvalVar` is called just like an ordinary static rule.

```
eval11 = rec e(
  try(App(e, e));
  try(EvalVar + (EvalApp + EvalLet); e)
)
```

Note that in this strategy the argument of an application is evaluated before it is associated with the abstraction variable, this ensures that only values are substituted for variables. Therefore, the result of applying `EvalVar` should not be evaluated recursively.

If the result of evaluation is not a first-order value, but rather contains functions, it is possible that those functions contain free variables that have a value assigned to it by a dynamic rule. Since evaluation does not proceed within abstractions, these free variables are not replaced with their values. Therefore the evaluation strategy `eval11` needs to be followed by a substitution phase that takes care of this.

```
subs11 = substitute(EvalVar)
```

The `substitute` strategy uses the dynamic rule `EvalVar` to replace free variables and at the same time renames bound variables to avoid variable capture.

## *4.2 Lazy Evaluation with Updating*

As was the case with the explicit substitution style, turning an eager evaluation strategy into a lazy evaluation strategy is a matter of adapting the traversal scheme.

```
eval12 = rec e(
  try(App(e, id));
  try((share(e) + (EvalApp + EvalLet)); e)
)
```

After evaluating a variable, its binding is updated such that subsequent evaluations of the variable are not evaluated again. The `share` strategy substitutes a variable, i.e., applies the dynamic rule `EvalVar`, then evaluates the expression, and finally updates the dynamic rule to yield this value the next time the variable is evaluated.

```
share(s) :
  [[ x ]] -> [[ e ]]
  where <EvalVar; s> [[ x ]] => [[ e ]]
      ; rules(EvalVar: [[ x ]] -> [[ e ]])
```

## 5 Pattern Matching

In the previous sections we have explored several styles of using rewriting strategies for the definition of interpreters for pure lambda expressions. Once a certain style has been chosen it becomes very easy to extend an interpreter with support for new language features. In this section and the next we consider the extension of lambda expressions with pattern matching and failure. We build on the dynamic rule style of evaluation, storing bindings in dynamically generated rules.

### *5.1 Rules*

First of all we extend the expression language with algebraic data type constructors. A term `Con(n)` is a data constructor with name `n`, which can be applied to a number of expressions using the ordinary application operator `App`. Thus, an example term with constructor applications is

```
App(App(Con("Plus"), Con("Zero")),
    App(Con("Succ"), Con("Zero")))
```

or in concrete syntax

```
Plus Zero (Succ Zero)
```

Values built from constructors can be taken apart by means of pattern matching rules. A rule is similar to an abstraction, but instead of just abstracting over a variable, it abstracts over a constructor application pattern. A rule consists of an expression on the left denoting a term pattern, and an expression on the right denoting the value of the rule. An example rule is

```
Plus Zero x -> x
```

or in abstract syntax

```
Rule(App(App(Con("Plus"), Con("Zero")), Var("x")), Var("x"))
```

The language of lambda expressions is extended with the `Con` and `Rule` constructors:

```
signature
  constructors
    Con  : String -> Exp     // Id -> Exp
    Rule : Exp * Exp -> Exp  // Exp "->" Exp -> Exp
```

### *5.2 Pattern Matching Reductions*

When applying a rule to a term, the left-hand side of the rule is matched against the term. If that succeeds, the instantiation of the right-hand side is produced. This can be expressed by reduction rules that decompose the pattern and subject term simultaneously.

If the pattern is a simple variable, it matches with any argument term, binding the argument to the variable using a new dynamic rule:

```
VarMatch :
  [[ (x -> e1) e2 ]] -> e1
  where rules(EvalVar : [[ x ]] -> [[ e2 ]])
```

A simple constructor only matches the same constructor:

```
ConMatch :
  [[ (c -> e) c ]] -> [[ e ]]
```

Complex terms and patterns are composed using the application operator. An application pattern matches an application value if the subterms match. This is achieved by decomposing the pattern and value applications into a curried rule and application:

```
AppMatch:
  [[ (p1 p2 -> e3) (e1 e2) ]] -> [[ (p1 -> p2 -> e3) e1 e2 ]]
```

Note that this application pattern match does not require a constructor at the bottom of the application spine; a pattern can be of the form `x y`, i.e., a variable can be used at the function position, and thus generically decompose a constructor application. This feature is introduced in RhoStratego [6] to support generic term traversal.

Note that this definition has a consequence for non-linear patterns, i.e., in which a variable occurs more than once. For example, the pattern in the left-hand side of the rule `(Plus x x) -> e3` contains two occurrences of the variable `x`. Matching this pattern against `Plus e1 e2`, eventually rewrites to

```
(x -> (x -> e3)) e1 e2
```

In other words, the inner `x` will shadow the outer one. Although this is different from the usual interpretation in which the arguments matched against these variables are required to be equal, this is a reasonable solution; in this setting it is not clear otherwise what the scope rules would be.

Finally, a let binding can be considered syntactic sugar for a rule application,

```
LetRule :
  [[ let x = e1 in e2 ]] -> [[ (x -> e2) e1 ]]
```

and abstraction is syntactic sugar for a simple rule

```
AbsRule :
  [[ \ x -> e ]] -> [[ x -> e ]]
```

## *5.3 Evaluation Strategy*

Based on these rules we can define a lazy evaluation strategy for the lambda calculus with lets and patterns, as follows:

```
rules13 = VarMatch + ConMatch + AppMatch + LetRule + AbsRule
```

```
  eval13 = rec e(
    try (App(e, id); try(App(Rule(not(Var(id)),id),e))
        + Rule(Var(id),id); rename);
    try(rules13; e + share(e))
  )
```

This strategy has a couple of interesting features. In lazy evaluation the argument of a rule application should not be evaluated, *unless* the argument is matched against a non-variable pattern. This is achieved by evaluating the argument only if the function reduces to a non-variable rule. Evaluation of the argument is done by a recursive call to the evaluation strategy. The main strategy reduces expressions to head normal form since arguments of applications are not evaluated. Furthermore, renaming of the bound variables in a rule is done in the strategy instead of in the rules.

## 6 Failure and Choice

The pattern matching extension presented in the previous section is only of limited value, since the definition of pattern matching did not include pattern match failure. A useful language with pattern matching must catch failure and provide a mechanism for choosing an alternative execution. Functional languages provide the case construct, which subsequently tries a number of rules until one succeeds. In RhoStratego the case construct has been taken apart into individual pattern matching rules and a choice operator. In this way, rules are first-class citizens that can be combined at will with other rules. Also pattern match failure itself becomes first-class. The conventional case construct can easily be defined in terms of rules and choice. In this section we further extend our language with failure and choice.

### *6.1 Pattern Match Failure*

First we extend the language with an explicit notion of failure. This is the special symbol that pattern match failure reduces to.

```
signature
  constructors
    Fail : Exp   // "fail" -> Exp
```

Rules `ConMatch` and `AppMatch` in the previous section will fail to apply when the application arguments are not the corresponding constructor or an application, respectively. This is expressed by the following negative reduction rules that reduce these cases to `fail`.

```
  ConMatchN:
    [[ (c -> e1) e2 ]] -> [[ fail ]]
    where <not(?[[ c ]])> [[ e2 ]]
  AppMatchN:
    [[ (p1 p2 -> e1) e2 ]] -> [[ fail ]]
    where <not(?[[ e3 e4 ]])> e2
```

Note that these negative rules only make sense in case the argument has been evaluated.

## *6.2 Failure Propagation and Choice*

When pattern match failure is made explicit it becomes possible to deal with it. We add a left-biased choice operator `e1 <+ e2` (`LChoice`), which first tries its first argument and if that results in failure, backtracks to its second argument. This operator makes it possible to choose between rules, but also between more complex computations that can result in failure.

```
signature
  constructors
    LChoice : Exp * Exp -> Exp // Exp "<+" Exp -> Exp
```

The choice operator is defined by the following reduction rules. If the left-hand side has `failed`, the choice reduces to the right-hand side. Otherwise, if the left-hand side is not `fail` (and not a (choice of) rule(s)), it reduces to the left-hand side.

```
  RightChoice :
    [[ fail <+ e ]] -> [[ e ]]

  LeftChoice :
    [[ e1 <+ e2 ]] -> [[ e1 ]]
    where
      <not(?[[ fail ]]
          + ?[[ e3 -> e4 ]]
          + ?[[ e5 <+ e6]])> e1
```

Again, these rules are only meaningful if the left-hand side of the choice is reduced to a value.

A choice with a rule as its left-hand side is not considered to succeed, yet. The choice is considered to be determined by the result of *application* of the rule. This is expressed by the following distribution rule, that distributes an application over a choice:

```
  Distrib:
    [[ (e1 <+ e2) e3 ]] -> [[ (e1 x) <+ (e2 x) ]]
    where new => x
        ; rules(EvalVar : [[ x ]] -> [[ e3 ]])
```

In order to share the computation associated with the argument expression, it is bound to a variable.

Failure propagates through applications:

```
  PropFunc:
    [[ fail e ]] -> [[ fail ]]

  PropArg:
    [[ e fail ]] -> [[ fail ]]
```

The `PropArg` rule is only relevant for eager strategies. In lazy strategies the value of the argument of an application is ignored, unless it is matched against.

A lazy evaluation strategy that includes pattern match failure, failure propagation and choice is a straightforward extension of `eval13`, adding the traversal of `LChoice` and the additional reduction rules:

```
  eval14 = rec e(
    try(App(e, id); try(App(Rule(not(Var(id)),id),e))
        + LChoice(e, id)
        + Rule(Var(id),id); rename);
    try(rules14; e + share(e))
  )
  rules14 =
    VarMatch + ConMatch + ConMatchN + AppMatch + AppMatchN
    + LeftChoice + RightChoice + PropFunc + Distrib
    + LetRule + AbsRule
```

An eager strategy can be derived from the lazy one by always evaluating function arguments and adding argument failure propagation.

This strategy provides an interpreter for a language with first-class rules and generic traversal. This is the basis of the RhoStratego language [6]. RhoStratego adds two extensions. Matching against failure is useful in a lazy evaluation setting to force evaluation. A cut operator that cuts off backtracking is useful to commit to a choice and prevent further applications of the `Distrib` rules.

## *6.3 Case is Sugar for Choice*

The traditional `case` construct of functional programming languages such as ML and Haskell can be implemented by means of rules and choice. It could be added to the language defined above as a convenience for the programmer by introducing the `case` construct, which uses a list of alternatives:

```
signature
  constructors
    Case : Exp * List(Alt) -> Exp
                   // "case" Exp "of" "{" {Alt ";"}* "}" -> Exp
    Alt  : Exp * Exp -> Alt  // Exp "->" Exp
```

This syntactic sugar can be desugared into an application of the choice of the alternatives, rendered as rules, to the case scrutinee:

```
  CaseDesugar :
    [[ case e1 of { as } ]] -> [[ e2 e1 ]]
    where
      <foldr(![[ fail ]]
            ,\ (e1, e2) -> [[ e1 <+ e2 ]] \
            ,\ [[ p -> e ]A] -> [[ p -> e ]] \ )> as => e2
```

Note that `foldr(s1,s2,s3)` is a combination of fold and map, i.e., reduces a list `[t1,...,tn]` to `<s2>(<s3>t1, <s2>(...,<s2>(<s3>tn, <s1>)))`. The anonymous rule `\ [[ p -> e ]A] -> [[ p -> e ]] \` transforms a case alternative into a rule. The `A` annotation in the left-hand side is used to disambiguate the term.

## 7 Concluding Remarks

The techniques in this paper have been applied in several realistic interpreters, including interpreters for Tiger, RhoStratego, and StrategoScript, an interpreter for Stratego specifications.

A specification of an interpreter for the Tiger language [1] is part of the Tiger-in-Stratego project [1] , which explores the application of program transformation techniques in compilation. The language includes imperative features such as loops, assignments, updatable records and arrays. Using dynamic rules these features can be expressed elegantly.

RhoStratego [6] is a functional programming language with built-in features for strategic rewriting. The language with pattern matching rules, choice and failure of Sections 5 and 6 forms the core of this language. A complete specification of the interpreter can be found in [6], its source is available online [2] .

Furthermore, we have experimented with the specification of optimizers for imperative programs (e.g., constant and copy propagation, dead code elimination) following the same approach; a set of optimization rules guided by a rewriting strategy using dynamic rules to transfer context-sensitive (data-flow) information. Many of the 'constant folding' rules from the interpreter can be reused in such optimizers. Although it is clear that an optimizer achieves a partial evaluation of a program, the relationship between interpreters and optimizers needs further investigation.

**Acknowledgements**

The lambda calculus evaluation strategies were first explored with Dick Kieburtz during a visit to INRIA Roquencourt in the Spring of 2000. The idea of using Stratego for specifying interpreters arose during a visit to INRIA/Lorraine in January 2001 during which occasion a prototype of an in-

[1] `http://www.stratego-language.org/Hpc/TigerInStratego`
[2] `http://www.stratego-language.org/Stratego/RhoStratego`

terpreter for the rho calculus was defined with Christophe Ringeissen and Pierre-Etienne Moreau. Karina Olmos implemented a number of strategies for imperative optimization. The LDTA referees provided constructive comments on the paper.

## References

[1] A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998. 18

[2] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. Elan: A logical framework based on computational systems. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996. Proceedings of the First Workshop on Rewriting Logic and Applications 1996 (WRLA'96). 2

[3] P. Borra, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. Technical Report 777, INRIA, Sophia Antipolis, 1987. 2

[4] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific, Singapore, September 1996. 2

[5] K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. In *Workshop on Language Descriptions, Tools and Applications (LDTA'2001)*, Genova, Italy, April 2001. 1, 2

[6] E. Dolstra. First-class rules and generic traversal for program transformation languages. Master's thesis, Utrecht University, Utrecht, The Netherlands, August 2001. 2, 3, 10, 14, 17, 18

[7] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. 2

[8] P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. 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. . 6

[9] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Passau, Germany, 1987. 2

[10] P. W. Kutter and A. Pierantonio. Montages specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416–442, May 1997. 1

[11] J. E. Labra Gayo, M. C. Luengo Díez, J. M. Cueva Lovelle, and A. Cernuda del Río. Lps: A language prototyping system using modular monadic semantics. In *Workshop on Language Definitions, Tools and Applications (LDTA'01)*, volume 44/2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science. 1

[12] G. L. Steele Jr. Building interpreters by composing monads. In *21st ACM Symp. on Principles of Programming Languages (POPL'94)*, pages 472–492, Portland, Oregon, January 1994. 1

[13] E. Visser. Scoped dynamic rewrite rules. In M. van den Brand and R. Verma, editors, *Rule Based Programming (RULE'01)*, volume 59/4 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001. 3, 11

[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. 3

[15] E. Visser. Meta-programming with concrete object syntax. In D. Batory and C. Consel, editors, *Generative Programming and Component Engineering (GPCE'02)*, Lecture Notes in Computer Science. Springer-Verlag, October 2002. (To appear). 3

[16] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998. 2, 3

[17] Y. Xiao, Z. M. Ariola, and M. Mauny. From syntactic theories to interpreters: A specification language and its compilation. In N. Dershowitz and C. Kirchner, editors, *Proceedings of the First International Workshop on Rule-Based Programming (RULE'00)*, Montréal, Canada, September 2000. `http://www.loria.fr/~ckirchne/=rule2000/proceedings/`. 2

[18] Y. Xiao, A. Sabry, and Z. M. Ariola. From syntactic theories to interpreters: Automating the proof of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4), 2001. 2