The Stratego Compiler

Specification of a Strategy Language

Eelco Visser

Department of Computer Science
Universiteit Utrecht

DRAFT
April 9, 1999

The Stratego Compiler

Specification of a Strategy Language

Eelco Visser

DRAFT
April 9, 1999

Abstract This report documents the specification of the compiler for the strategy language Stratego. The compiler is specified in Stratego itself.

Address:
Department of Computer Science
Universiteit Utrecht
PO Box 80089
3508 TB Utrecht
email: visser@acm.org
http://www.cs.uu.nl/~visser/

Contents

Chapter 1

A Bootstrapped Compiler for Strategies

Stratego is a language for the specification of program transformation based on rewriting strategies. The Stratego compiler is based on program transformation; it transforms a high-level Stratego specification via several intermediate representations to C. Several optimizations are performed on the intermediate representations. The compiler is bootstrapped, i.e., it is specified in Stratego itself. In this paper we give an overview of the Stratego compiler: architecture; issues in the compilation of strategies; some high-lights of the specification; and experience with using strategies for writing program transformations.

1.1 Introduction

One of the shortcomings of implementing algebraic specifications by means of term rewriting is the necessity to encode the strategy to apply 'equations' by means of functions. This obscures the equational nature of specifications and hampers their modularity because rules become part of one particular strategy. This observation was done in several projects that used ASF+SDF [3] to define language processors, in particular program transformations. For example: normalization of box expressions for pretty-printing [2]; normalization of a syntax definitions [8]; transformation of C++ programs [4].

Stratego is a language for the specification of program transformations that overcomes this shortcoming by providing user-definable rewriting strategies. A rewriting strategy is an expression in a language of strategy operators that combine rules (usually via their labels) in a program that traverses a term and applies the rules.

The preliminary ideas for the strategy operators in Stratego, inspired by the specification formalism ELAN [1], include operators for sequential nondeterministic programming, data-type specific and generic term traversal. In [6] an implementation of these operators in ASF+SDF is described. That interpretive implementation style can be used to write strategies for ASF+SDF equations and can be applied in other settings as well; the traditional way of

defining a strategy with functions is replaced by a style in which an evaluation function interprets a strategy expression. This allows the concise specification of various strategies and the use of one rule with many different strategies.

These preliminary ideas are further developed in [10, 11]. Rewrite rules are no longer primitives, but are broken down into operations for matching, building and variable scope. System S, the resulting set of strategy operators, provides the primitive operations for defininig both rewrite rules and strategies for applying them. A Stratego specification provides syntactic abstractions on top of System S. For example, a rewrite rule is an abstraction for a sequence of operations that first matches the subject term against a pattern, then satisfies a condition and finally builds the instantiation of a term pattern. The identification of this intermediate level allows the definition of very expressive abstractions such as contextual rules and overlays [9].

The Stratego compiler first translates a high-level Stratego specification to a System S expresssion, which is then translated to a list of abstract machine instructions that are implemented in C. The first compiler was written in SML. Based on the first experience with that compiler an improved compiler was specified in Stratego itself and bootstrapped. Bootstrapping proves to be a good strategy for developing the compiler and the language because it provides a realistic case study and a good test case for the compiler. The compiler is being used in several program transformation case studies, such as a specificiation of an optimizer for a functional language [11], a deforestation algorithm for a functional language [5] and a transformation tool to speed up C++ programs for high-performance computing.

In this abstract we give an overview of System S and Stratego, present the architecture of the compiler and show some examples of the use of strategies in the compiler. In the full paper we will further elaborate the application of strategies in the compiler and give a first evaluation of the use of strategies in program transformation.

1.2 The Language

This section introduces System S, a calculus for the definition of tree transformations, and Stratego, a specification language providing syntactic abstractions for System S expressions. For an operational semantics see [10, 11].

1.2.1 System S

System S is a hierarchy of operators for expressing term transformations. The first level provides control constructs for sequential non-deterministic programming, the second level introduces combinators for term traversal and the third level defines operators for binding variables and for matching and building terms.

First-order terms are expressions over the grammar

```
t := x | C(t1,...,tn) | [t1,...,tn] | (t1,...,tn)
```

where `x` ranges over variables and `C` over constructors. The arity and types of constructors are declared in signatures. The notation `[t1,...,tn]` abbreviates the list `Cons(t1,...,Cons(tn,Nil))`. Transformations in System S are applied to ground terms, i.e., terms withouth variables.

```
module traversals
imports lists
strategies
  try(s)    = s <+ id              map(s)    = rec x(Nil + Cons(s, x))
  repeat(s) = rec x(try(s; x))     list(s)   = rec x(Nil + Cons(s, x))
  topdown   = rec x(s; all(x))     alltd(s)  = rec x(s <+ all(x))
  bottomup  = rec x(all(x); s)     oncetd(s) = rec x(s <+ one(x))
  downup(s) = rec x(s; all(x); s)  sometd(s) = rec x(s <+ some(x))
  onebu(s)  = rec x(one(x) <+ s)   somebu(s) = rec x(some(x) <+ s)
  downup2(s1, s2) = rec x(s1; all(x); s2)
```

Figure 1.1: Specification of several generic term traversal strategies.

Level 1: Sequential Non-deterministic Programming Strategies are programs that attempt to transform ground terms into ground terms, at which they may succeed or fail. In case of success the result of such an attempt is a transformed term. In case of failure the result is an indication of the failure. Strategies can be combined into new strategies by means of the following operators: The *identity* strategy `id` leaves the subject term unchanged and always succeeds. The *failure* strategy `fail` always fails. The *sequential composition* `s1; s2` first attempts to apply `s1` to the subject term and, if that succeeds, applies `s2` to the result. The *non-deterministic choice* `s1 + s2` attempts to apply either `s1` or `s2`. It succeeds if either succeeds and it fails if both fail; the order in which `s1` and `s2` are tried is unspecified. The *deterministic choice* `s1 <+ s2` attempts to apply either `s1` or `s2`, in that order. The *recursive closure* `rec x(s)` attempts to apply `s`, where at each occurence of the variable `x` in `s`, the strategy `rec x(s)` is applied. The *test* strategy `test(s)` tries to apply `s`. It succeeds if `s` succeeds, and reverts the subject term to the original term. It fails if `s` fails. The *negation* `not(s)` succeeds (with the identity transformation) if `s` fails and fails if `s` succeeds. Two examples of strategies defined with these operators are `try` and `repeat` in Figure 1.1.

Level 2: Term Traversal The Level 1 constructs apply transformations to the root of a term. In order to apply transformations throughout a term it is necessary to traverse it. For this purpose, System S provides the following operators: For each `n`-ary constructor `C` the *congruence* operator `C(s1,...,sn)` is defined. It applies to terms of the form `C(t1,...,tn)` and applies `si` to `ti` for `1 <= i <= n`. An example of the use of congruences is the operator `map(s)` in Figure 1.1 that applies `s` to each element of a list.

Congruences can be used to define traversals over specific data structures. Specification of generic traversals (e.g., pre- or post-order over arbitrary structures) requires more generic operators. The operator `all(s)` applies `s` to all children of a constructor application `C(t1,...,tn)`. In particular, `all(s)` is the identity on constants (constructor applications without children). The strategy `one(s)` applies `s` to one child of a constructor application `C(t1,...,tn)`; it is precisely the failure strategy on constants. The strategy `some(s)` applies `s` to some of the children of a constructor application `C(t1,...,tn)`, i.e., to at least one and as many as possible. Like `one(s)`, `some(s)` fails on constants.

Figure 1.1 defines various traversals based on these operators. For instance, `oncetd(s)` tries to find *one* application of `s` somewhere in the term starting at

the root working its way down; `s <+ one(x)` first attempts to apply `s`, if that fails an application of `s` is (recursively) attempted at one of the children of the subject term. If no application is found the traversal fails. Compare this to the traversal `alltd(s)`, which finds *all* outermost applications of `s` and never fails.

Level 3: Match, Build and Variable Binding The operators introduced thus far are useful for repeatedly applying transformation rules throughout a term. Actual transformation rules are constructed by means of pattern matching and building of pattern instantiations.

A match `?t` succeeds if the pattern term `t` matches the subject term. As a side-effect, any variables in `t` are bound to the corresponding subterms of the subject term. If a variable was already bound before the match, then the binding only succeeds if the terms are the same. This enables non-linear pattern matching, so that a match such as `?F(x, x)` succeeds only if the two arguments of `F` in the subject term are equal. This non-linear behaviour can also arise accross other operations. For example, the two consecutive matches `?F(x, y); ?F(y, x)` succeed exactly when the two arguments of `F` are equal. Once a variable is bound it cannot be unbound.

A build `!t` replaces the subject term with the instantiation of the pattern `t` using the current bindings of terms to variables in `t`. A scope `{x1,...,xn: s}` makes the variables `xi` local to the strategy `s`. This means that bindings to these variables outside the scope are undone when entering the scope and are restored after leaving it. The operation `where(s)` applies the strategy `s` to the subject term. If successful, it restores the original subject term, keeping only the newly obtained bindings to variables.

1.2.2 Stratego

The specification language Stratego provides syntactic abstractions for System S expressions. A specification consists of a collection of modules that define signatures, transformation rules and strategy definitions.

A signature declares the sorts and operations (constructors) that make up the structure of the language(s) being transformed. Example signatures are shown in Figures 1.2 and 1.3. A strategy definition `f(x1,...,xn) = s` introduces a new strategy operator `f` parameterized with strategies `x1` through `xn` and with body `s`. Such definitions cannot be recursive, i.e., they cannot refer (directly or indirectly) to the operator being defined. All recursion must be expressed explicitly by means of the recursion operator `rec`. Labeled transformation rules are abbreviations of a particular form of strategy definitions. A conditional rule `L : l -> r where s` with label `L`, left-hand side `l`, right-hand side `r`, and condition `s` denotes a strategy definition `L = {x1,...,xn: ?l; where(s); !r}`. Here, the body of the rule first matches the left-hand side, and then attempts to satisfy the condition `s`. If that succeeds, then it builds the right-hand side `r`. The rule is enclosed in a scope that makes all term variables `xi` occurring in `l`, `s` and `r` local to the rule. If more than one definition is provided with the same name, e.g., `f(xs) = s1` and `f(xs) = s2`, this is equivalent to a single definition with the sum of the original bodies as body, i.e., `f(xs) = s1 + s2`.

The following definitions provide a useful shorthand. The notation `<s> t` denotes `!t; s`, i.e., the strategy that builds the term `t` and then applies `s` to it. The notation `s => t` denotes `s; ?t`, i.e., the strategy that applies `s` to the

```
module terms
signature
  sorts Term
  operations
    Wld :                          Term (* _ *)
    Var : String                -> Term (* x *)
    Int : Int                   -> Term (* 0, 1, 2 ... *)
    Str : String                -> Term (* "", "a", ... *)
    Op  : String * List(Term) -> Term (* f(t1,...,tn) *)
```

Figure 1.2: Abstract syntax of terms.

current subject term and then matches the result against `t`. The combined notation `<s> t => t'` thus denotes `(!t; s); ?t'`. The `<s> t` notation can also be used in a build expression. For example, the strategy expression `!F(<s> t, t')` corresponds to `{x: <s> t => x; !F(x,t')}`, where `x` is a new variable.

1.2.3 Library

The language comes with a growing library of strategy operators with functionality for

- Simple traversals such as in Figure 1.1
- Fixed-point traversals that keep applying transformations
- List operations
- Tuple operations
- Manipulation of expressions with (bound) variables, such as variable renaming, substitution, collection of the set of free variables etc. These operations are language independent and can be specialized to a language by instantiation of generic operations.

1.3 The Compiler

A Stratego specification defines a transformation on terms. The Stratego compiler translates a specification to an executable program that reads in a term, transforms it according to the specification and outputs the resulting transformed term. In this section we discuss the architecture of the compiler and the run-time system used in the generated programs. In the next section we give some examples of the specification of the compiler in Stratego itself.

1.3.1 Architecture

The overall architecture of the compiler is shown in Figure 1.4. The compiler consists of four main components: front-end, optimizer, matching-tree, and back-end.

```
module strategy
imports terms
imports list-cons
signature
  sorts SVar Strat SDef
  operations
    Id      :                                   Strat (* id *)
    Fail    :                                   Strat (* fail *)

    Test    : Strat                          -> Strat (* test s *)
    Not     : Strat                          -> Strat (* not s *)

    Seq     : Strat * Strat                  -> Strat (* s1 .  s2 *)
    Choice  : Strat * Strat                  -> Strat (* s1 +  s2 *)
    LChoice : Strat * Strat                  -> Strat (* s1 <+ s2 *)

    SVar    : String                         -> SVar
    Rec     : String * Strat                 -> Strat (* rec x . s *)
    Let     : SDef * Strat                   -> Strat (* let sdef in s2*)
    SDef    : String * List(String) * Strat -> SDef  (* f(xs) = s *)
    Call    : SVar * List(Strat)             -> Strat (* f(ss) *)

    Path    : Int * Strat                    -> Strat (* i(s) *)
    Cong    : String * List(Strat)           -> Strat (* f(s1,...,sn) *)
    One     : Strat                          -> Strat (* one(s) *)
    Some    : Strat                          -> Strat (* some(s) *)
    All     : Strat                          -> Strat (* all(s) *)

    Match    : Term                          -> Strat (* match(s) *)
    Build    : Term                          -> Strat (* build(s) *)

    Scope   : List(String) * Strat           -> Strat (* {xs: s} *)
    Where   : Strat                          -> Strat (* where s *)

    Prim    : String -> Strat
```

Figure 1.3: Abstract syntax of System S expressions.

The front-end takes a Stratego specification (in abstract syntax form) and translates it to a list of System S expressions (SSE). The front-end itself is composed of five stages: joining sections of the same kind (normalization); translation of rules and signatures to strategy definitions; extraction of the definitions that are actually needed for implementation of the operator `main`; elimination of the syntactic abstractions (sugar) of the Stratego level; and (selective) inlining definitions. The result of this stage is a list of strategy definitions for parameterless operators.

The optimizer simplifies a System S expression by applying algebraic laws. The matching-tree automaton transformer targets expressions that are choices of strategies starting with a match. Common prefixes are extracted to prevent inspecting a term more than once. The back-end translates an expression to abstract machine instructions.

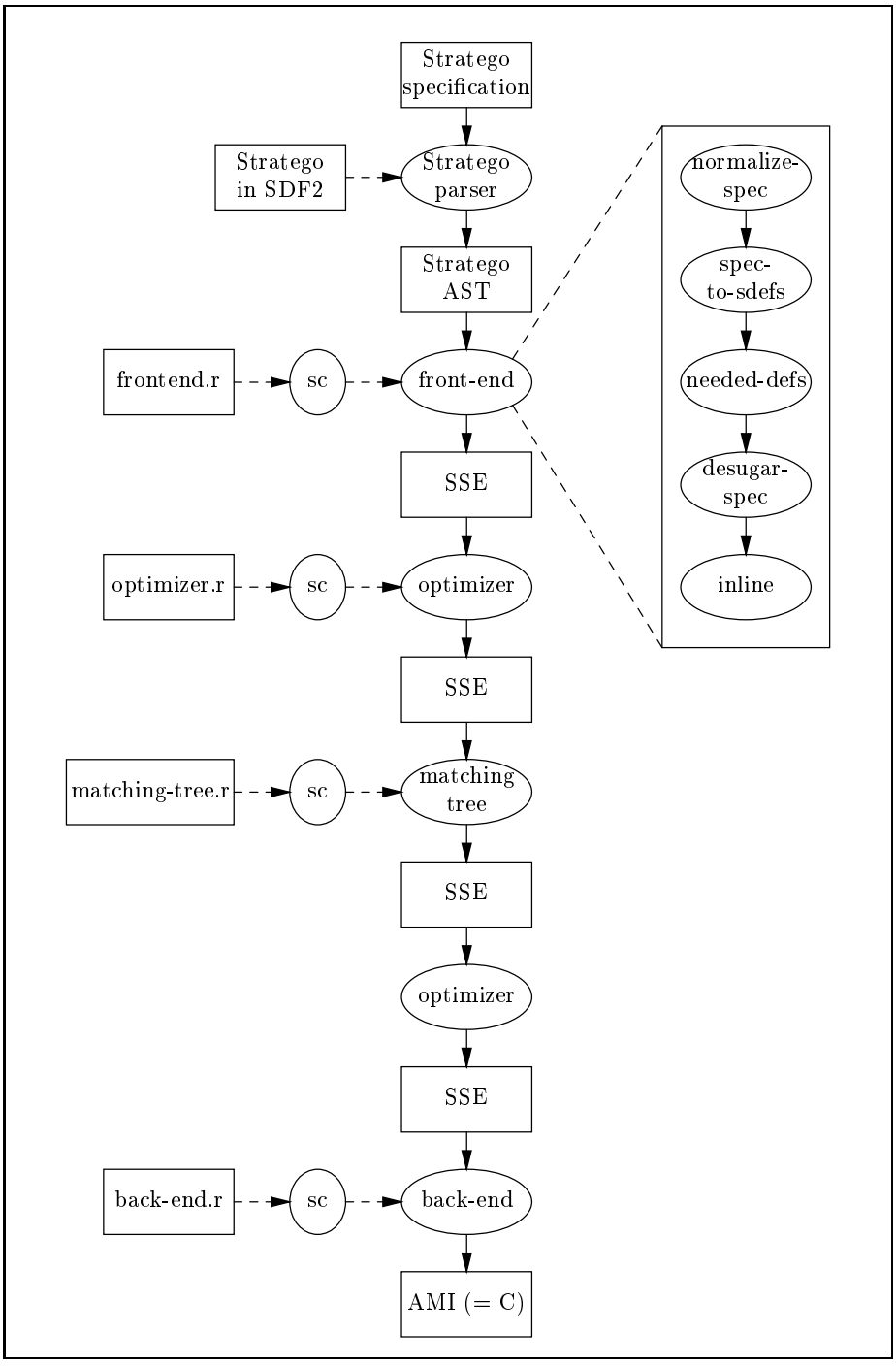

Figure 1.4: Architecture of the Stratego compiler (sc)

1.3.2 Run-time System

The abstract machine instructions produced by the compiler are implemented as macros or procedures in C. These procedures make use of a run-time system that supports stack and term management. The return- and choice-stacks are needed for control. The term-stack is used to deconstruct terms in matching and term traversal. For the creation and garbage collection of terms the ATerms package [7] is used.

The run-time system is implemented in C. Therefore, the compiler and generated transformation programs only depend on `gcc`. The Stratego compiler (sc) is being developed on a Linux platform and is also used on Sun machines. Although this has not been done, there should be no problem in porting the compiler to Windows NT platforms with GNU software.

1.4 Examples

In this section we give some examples to illustrate the specification of the compiler in Stratego.

1.4.1 Example 1: Pipelines

The `main` strategy of the front-end defines a pipeline of operations. In addition to the operations discussed in the previous section, use-def analysis is used to determine if variables are used in builds without being bound in match operations.

```
strategies

  main = normalize-spec;
         where(spec-use-def);
         spec-to-sdefs;
         needed-defs;
         desugar-spec;
         inline
```

1.4.2 Example 2: Desugaring

The following specification is a fragment from module `desugar.r` that defines the elimination of syntactic abstractions (sugar). Rules `Bapp2` transforms a build expression such as `!F(<s> G(y))` to `{x: <s> G(y) => x; !F(x)}` in order to extract the strategy application inside the build.

Rule `Bapp2` uses a contextual pattern `t[App(s, t')]` to reach an application `App(s, t')` at an arbitrary depth inside the term matching `t`. This application is replaced with the newly generated variable `Var(x)` in the right-hand side by means of the context `t[Var(x)]`.

The desugaring strategy `desugar` desugars an expression by applying a set of rules, including `Bapp2`, repeatedly in a topdown traversal. The strategy `desugar-spec` applies this strategy to each definition body in a list of definitions.

```
rules

  Bapp2 : Build(t[App(s, t')]) ->
          Scope([x], Seq(BAM(s, t', Var(x)), Build(t[Var(x)])))
          where new => x

strategies

  desugar = topdown(try(desugarRule);
                    repeat(HL + (Bapp0 <+ Bapp1 <+ Bapp2)))

  desugar-spec = map(SDef(id, id, desugar))
```

1.4.3 Example 3: Compilation

The back-end of the compiler translates expressions to lists of abstract machine instructions. For each language construct a rule defines the pattern of instructions it corresponds to. For example, the following rules define the translation of left choice `<+` and the generic traversal operator `all`.

```
rules
  C : Instr(LChoice(s1, s2), env, rcs) ->
      Block([Cpush(fc),
             Instr(s1, env, rcs), Cpop, Goto(sc),
             Label(fc), Instr(s2, env, rcs),
             Label(sc)])
      where new => sc ; new => fc

  C : Instr(All(s), env, rcs) ->
      Block([AllInit,
             Label(c1),
             AllNextSon(c2),
             Instr(s, env, rcs),
             Goto(c1),
             Label(c2),
             AllBuild])
      where new => c1 ; new => c2
```

These rules are combined in the compilation strategy `compile`. The default `C` rules are applied after the specialized `Cs` rules.

```
strategies

  compile = map(MkInstr;
                topdown(repeat(Cs <+ C)));
            Assemble;
            flatten-blocks;
            peephole
```

Chapter 2

Syntax

2.1 Terms

Terms are either variables, integers, strings or applications of a function symbol to a list of terms. A wildcard corresponds to an anonymous variable. In comments after the declarations of the operators an example of the notation used in concrete syntax.

```
module terms
signature
  sorts Term
  operations
    Wld :                          Term   (* _ *)
    Var : String                -> Term   (* x *)
    Int : Int                   -> Term   (* 0, 1, 2 ... *)
    Str : String                -> Term   (* "", "a', ... *)
    Op  : String * List(Term) -> Term   (* f(t1,...,tn) *)
```

Remarks:

List notation [t1,...,tn] translates to Cons(t1, ..., Cons(tn, Nil))

Tuple notation (t1,...,tn) translates to TCons(t1, ..., TCons(tn, TNil))

2.2 Strategies

In this module we define the operators of the core strategy language that will be used to express all programs in the high-level language.

```
module strategy
imports terms
imports list-cons
```

Sequential Non-deterministic Programs The following operators provide a language for sequential non-deterministic programming.

```
signature
  sorts SVar Strat SDef
  operations
    Id      :                                  Strat    (* id *)
    Fail    :                                  Strat    (* fail *)

    Test    : Strat                         -> Strat    (* test s *)
    Not     : Strat                         -> Strat    (* not s *)

    Seq     : Strat * Strat                 -> Strat    (* s1 .  s2 *)
    Choice  : Strat * Strat                 -> Strat    (* s1 +  s2 *)
    LChoice : Strat * Strat                 -> Strat    (* s1 <+ s2 *)

    SVar    : String                        -> SVar
    Rec     : String * Strat                -> Strat    (* rec x . s *)
    Let     : SDef * Strat                  -> Strat    (* let sdef in s2 *)
    SDef    : String * List(String) * Strat -> SDef     (* f(xs) = s *)
    Call    : SVar * List(Strat)            -> Strat    (* f(ss) *)
```

Traversal Operators

```
    Path    : Int * Strat                   -> Strat    (* i(s) *)
    Cong    : String * List(Strat)          -> Strat    (* f(s1,...,sn) *)
    One     : Strat                         -> Strat    (* one(s) *)
    Some    : Strat                         -> Strat    (* some(s) *)
    All     : Strat                         -> Strat    (* all(s) *)
    Kids    :                                  Strat    (* kids *)
```

Matching and Building

```
signature
  operations
    Match    : Term                          -> Strat    (* match(s) *)
    Build    : Term                          -> Strat    (* build(s) *)
    MatchVar : String                        -> Strat    (* matchv(x) *)
    MatchFun : String                        -> Strat    (* matcho(f) *)

    Scope    : List(String) * Strat          -> Strat    (* {xs: s} *)
    Where    : Strat                         -> Strat    (* where s *)
```

Annotation

```
    Mark    : Term                           -> Strat    (* ! t *)
    IsMark  : Term                           -> Strat    (* ? t *)

    AnnBuild  : Term * Term                  -> Strat
```

```
    AnnMatch  : Term * Term                -> Strat
    AnnRemove : Term                       -> Strat
```

Primitives Example primitives for Integers are: Plus, Minus, Geq, NewInt (generate new integer), etc. *)

Strings New, StrConc

```
signature
  operations
    Prim    : String -> Strat

    CountRule : String -> Strat
```

2.3 Signatures

Types and Kinds

```
module signatures
signature
  operations
    ConstType : Term              -> Type
    FunType   : List(Term) * Term -> Type
```

Signatures

```
signature
  sorts SortDecl OpDecl BSig
  operations
    Sort       : String * List(Kind) -> SortDecl
    OpDecl     : String * Type       -> OpDecl
    Sorts      : List(SortDecl)      -> BSig
    Operations : List(OpDecl)        -> BSig
```

2.4 Syntactic Sugar

In this section we define specifications that declare signatures, rules and strategy definitions. This is just syntactic sugar for declaring a strategy. In Section **??** we will show how these constructs can be defined in terms of the basic strategies of Section 1.2.

```
module sugar
imports strategy
imports signatures
```

Variadic versions of some combinators

```
signature
  operations
    Seqs     : List(Strat) -> Strat
    Choices  : List(Strat) -> Strat
    LChoices : List(Strat) -> Strat
```

Match, Build, Apply Combinations

```
signature
  operations
    MA  : Term  * Strat       -> Strat   (* t => s *)
    AM  : Strat * Term        -> Strat   (* s => t *)
    BA  : Strat * Term        -> Strat   (* <s> t *)
    BAM : Strat * Term * Term -> Strat   (* <s> t => t' *)
```

Contexts & Term Application

```
signature
  operations
    Con : Var * Term * SVar -> Term    (* x[t](f) *)
    App : Strat * Term      -> Term    (* <s> t *)
```

Rules & Rule Definitions

```
signature
  sorts Rule RDef
  operations
    Rule  : Term * Term * Strat          -> Rule    (* t1 -> t2 where s *)
    SRule : Rule                         -> Strat
    RDef  : String * List(String) * Rule -> RDef    (* f(xs) : r *)
```

Specifications

```
signature
  sorts BSpec Spec
  operations
    Signature     : List(BSig)  -> BSpec
    Rules         : List(RDef)  -> BSpec
    Strategies    : List(SDef)  -> BSpec
    Specification : List(BSpec) -> Spec
```

2.5 Library for Strategies

This module instantiates several language independent functions defined in module subs to the strategy language.

```
module stratlib
imports strategy
imports subs

rules

  Add1 : Var(x)  -> [x]
  Add2 : SVar(x) -> [x]

  IsVar  : Var(x) -> x
  IsSVar : Call(SVar(x), []) -> x
  MkTVar : x -> Var(x)
  MkSVar : x -> SVar(x)
  MkCall : x -> Call(SVar(x), [])

  Bind0 : Scope(xs, s) -> xs
  Bind1 : Let(SDef(f, xs, s1), s2) -> [f]
  Bind2 : SDef(f, xs, s) -> xs
  Bind3 : Rec(x, s) -> [x]

strategies

  tvars = vars(Add1, Bind0)
  svars = vars(Add2, Bind1 + Bind2 + Bind3)

  tsubs = subs(IsVar)
  ssubs = subs(IsSVar)

  rn_apply(nvs, bnd, nbnd) = Scope(nvs, bnd)
                           + Let(SDef(nvs ; Hd, id, id) ; nbnd, bnd)
                           + SDef(id, nvs, bnd)
                           + Rec(nvs ; Hd, bnd)

  trename = rename(Var(id), MkTVar, Bind0)
  srename = rename(SVar(id), MkSVar, Bind1 + Bind2 + Bind3)

  strename = trename ; srename

  is_var_list = map({x:match(Var(x))})
  is_svar_list = map({x:match(SVar(x))})
```

Context strategies for strategies

```
strategies

  conLChoice(s) = rec x(s + LChoice(x, id) + LChoice(id, x))

  conChoice(s) = rec x(s + (Choice(x, id) + Choice(id, x)))

  conChoiceL(s) = Choice(s, id) + s
```

```
  choicebu-l'(s) = rec x(try(Choice(id, x); s))

  choicebu-l(s) = rec x(try(Choice(x, x); s))

  choicetd(s) = rec x(s <+ Choice(x, x))

  choicemap(s) = rec x(Choice(x, x) <+ s)

  choicebu(s) = rec x(try(Choice(x, x); s))

  firstInSeq(s) = s <+ Seq(s, id)

  lastInSeq(s) = Seq(id, rec x(s <+ Seq(not(oncetd(s)), x)))
```

2.6 L0

```
module L0
imports strategy
strategies

  string = id
  int = id

  basic-term(x) =
    Var(string)
  + Int(int)
  + Str(string)
  + Op(string, map(x))

  basic-strat(x) =
    Id
  + Fail
  + Seq(x, x)
  + LChoice(x, x)
  + Choice(x, x)
  + Not(x)
  + Test(x)
  + Scope(map(string), x)
  + Rec(string, x)
  + Let(SDef(string, map(string), x), x)
  + Cal(SVar(string), map(x))
  + Path(int, x)
  + Cong(string, map(x))
  + One(x)
  + Some(x)
  + All(x)
  + Kids
```

```
  + Where(x)
  + Prim(string)
  + CountRule(string)
```

L0 Terms have contexts and applications

```
strategies

  m-term = rec x(term(x))

  mc-term = rec x(
    basic-term(x)
  + Con(Var(string), x, string)
  )

  b-term(strat) = rec x(
    basic-term(x)
  + App(strat, x)
  )

  bc-term(strat) = rec x(
    basic-term(x)
  + Con(Var(string), x, string)
  + App(strat, x)
  )

  L0-strat = rec x(
    basic-strat(x)
  + BA(x, b-term(x))
  + AM(x, m-term(x))
  + Match(m-term(x))
  + Build(b-term(x))
  + SRule(Rule(mc-term, bc-term(x), x))
  )

  type = FunType(map(id), id) + ConstType(id)

  sig = Signature(map(Sorts(map(Sort(string, id))) +
                      Operations(map(OpDecl(string, type)))))

  rdef = RDef(string, map(string), Rule(term(strat), term(strat)
  rdefs = Rules(map(rdef))

  section = sig + rdefs + sdefs

  L0 = Specification(map(section))
```

Chapter 3

Frontend

In this section we define the desugaring procedure that translates a specification in the high-level syntax to an expression in the core language.

3.1 Frontend

```
module frontend
imports normalize-spec spec-to-sdefs needed-defs desugar inlining use-def
strategies

  main = normalize-spec;
         where(spec-use-def);
         spec-to-sdefs;
         needed-defs;
         desugar-spec;
         inline
```

3.2 NORMALIZE-SPEC

```
module normalize-spec
imports strategy
rules

  Names   : Signature(bsigs) -> <map(OpNames <+ ![]); concat> bsigs
  OpNames : Operations(ods)  -> <map(OpName <+ ![]); concat> ods
  OpName  : OpDecl(f, ConstType(_)) -> [f]

  Triple : ((x, y), z) -> (x, y, z)

strategies

  vars-to-consts = Dupl;
```

```
                  (map(Names <+ ![]);
                   concat;
                   Dupl; (id, map({x: ?x; !Op(x,[])}))
                  ,id);
                  Triple;
                  tsubs

rules

  BSpecs : Specification(bspecs) -> bspecs

  NormBSIG : Operations(ods) -> ods
  NormBSIG : Sorts(ss) -> []

  NormBSP : Signature(bsigs)  -> (<normalize-sigs> bsigs, [], [])
  NormBSP : Strategies(sdefs) -> ([], [], sdefs)
  NormBSP : Rules(rdefs)      -> ([], rdefs, [])

  Combine : ((ods1, rdefs1, sdefs1), (ods2, rdefs2, sdefs2)) ->
            (<conc> (ods1, ods2),
             <conc> (rdefs1, rdefs2),
             <conc> (sdefs1, sdefs2))

  MkSpec : (ods, rdefs, sdefs) ->
           Specification([Signature([Operations(ods)]),
                          Rules(rdefs),
                          Strategies(sdefs)])

strategies

  normalize-sigs =
        map(NormBSIG);
        concat

  normalize-spec =
        BSpecs;
        map(NormBSP);
        foldr(!([], [], []), Combine);
        MkSpec;
        Specification(vars-to-consts)
```

3.3 USE-DEF

```
module use-def
imports sugar list-set stratlib
strategies

  default-join = kids; tfoldr(![([], [], [])], cart(Union))
```

```
  use-def = rec x((constructs(x) <+ all(x)); (UDjoin <+ default-join))

rules

  Union : ((u1, d1, e1), (u2, d2, e2)) ->
          (<union> (u1, u2), <union> (d1, d2), <union> (e1, e2))

  SeqUnion : ((u1, d1, e1), (u2, d2, e2)) ->
             (<union> (u1, <diff> (u2, d1)),
              <union> (d1, d2), <union> (e1, e2))

strategies

  seq-join = cart(SeqUnion)
  seqs-join = foldr(![[], [], []], cart(SeqUnion))

rules

  UDjoin : Seqs(xs)       -> <seqs-join> xs
  UDjoin : Seq(xs, ys)    -> <seqs-join> [xs, ys]
  UDjoin : Rule(l, r, c)  -> <seqs-join> [l, c, r]
  UDjoin : StratRule(l, r, c) -> <seqs-join> [l, c, r]
  UDjoin : MA(t, s)       -> <seqs-join> [t, s]
  UDjoin : AM(s, t)       -> <seqs-join> [s, t]
  UDjoin : BA(s, t)       -> <seqs-join> [t, s]
  UDjoin : BAM(s, t1, t2) -> <seqs-join> [t1, s, t2]

  UDjoin : Scope(xs, uds) -> <map(JoinScope(!xs))> uds

  JoinScope(xs) : (u, d, e) ->
                  (u, <diff> (d, <xs>()),
                      <conc> (<isect> (u, <xs>()), e))

  MsgU : Nil -> []
  MsgU : Cons(x, Nil) ->
         ["variable ", x, ": used, but not bound"]
  MsgU : Cons(x, Cons(y, ys)) ->
         ["variables ", Cons(x, Cons(y, ys)), ": used, but not bound"]

  MsgD : Nil -> []
  MsgD : Cons(x, Nil) ->
         ["variable ", x, ": matched, but not declared"]
  MsgD : Cons(x, Cons(y, ys)) ->
         ["variables ", Cons(x, Cons(y, ys)), ": matched, but not declared"]

  MsgE : Nil -> []
  MsgE : Cons(x, Nil) ->
         ["variable ", x, ": declared, but not bound"]
  MsgE : Cons(x, Cons(y, ys)) ->
         ["variables ", Cons(x, Cons(y, ys)), ": declared, but not bound"]
```

```
  MsgS : (u, d, e) -> <concat> [<MsgU> u, <MsgD> d, <MsgE> e]
  MsgR : (u, d, e) -> <concat> [<MsgU> u, <MsgE> e]

  err-msg : RDef(l, xs, uds) ->
            Error(<conc> (["in rule ", l, " : "],
                          <map(MsgR)> uds))

  err-msg : SDef(f, xs, uds) ->
            Error(<conc> (["in definition ", f, " : "],
                          <map(MsgS); concat> uds))

strategies

  check = SDef(id, id, use-def; not([([], [], [])])); err-msg +
          RDef(id, id, use-def; not([([], id, [])])); err-msg


  use-term = {t: ?t; ![(<tvars> t, [], [])]}
  def-term = {t: ?t; ![([], <tvars> t, [])]}

  constructs(x) =
         Build(use-term) + Match(def-term) + MA(def-term, x) +
         AM(x, def-term) + BA(x, use-term) + BAM(x, use-term, def-term) +
         Scope(id, x) + Rule(def-term, use-term, x)

  defs-use-def = filter(check); ?[]

  spec-use-def = Specification([Signature(id),
                                Rules(defs-use-def),
                                Strategies(defs-use-def)])
```

3.4 Specification to List of Definitions

Translation of a specification consisting of a signature, rules and strategy definitions to a list of strategy definitions.

```
module spec-to-sdefs
imports strategy
imports sugar
imports stratlib
imports list-sort
```

Congruences Congruences are recognized by the parser as strategy calls; The following strategy generates strategy definitions from the signature; For instance, the operator declaration

```
OpDecl("F", FunType([_, _], _))
```

is translated to the strategy definition

```
    SDef("F", ["x1", "x2"], Cong("F", [SVar("x1"), SVar("x2")]))

rules

  MkCongDef : OpDecl(f, ConstType(t)) -> SDef(f, [], Cong(f, []))

  MkCongDef : OpDecl(f, FunType(ts, t)) -> SDef(f, xs, Cong(f, <map(MkCall)> xs))
              where <map(new)> ts => xs

  MkCongDefs : Sorts(sds)      -> []
  MkCongDefs : Operations(ods) -> <map(MkCongDef)> ods

strategies

  congdefs = map(MkCongDefs); concat
```

Rule Definitions to Strategy Definitions A rule definition defines an implicitly scoped strategy definition;

```
rules

  RDtoSD : RDef(f, xs, r) -> SDef(f, xs, Scope(<tvars> r, SRule(r)))
```

Primitives

```
  AddPrimitives : sdefs -> Cons(l, sdefs)
  where
  build([
    SDef("minus", [], Prim("minus")),
    SDef("plus",  [], Prim("plus")),

    SDef("add",   [], Prim("add")),
    SDef("subt",  [], Prim("subt")),
    SDef("mul",   [], Prim("mul")),
    SDef("div",   [], Prim("div")),

    SDef("geq",   [], Prim("geq")),
    SDef("leq",   [], Prim("leq")),
    SDef("gt",    [], Prim("gt")),
    SDef("lt",    [], Prim("lt")),

    SDef("implode-string", [], Prim("implode_string")),
    SDef("explode-string", [], Prim("explode_string")),

    SDef("mkterm", [], Prim("mkterm")),

    SDef("read",  [], Prim("read")),
    SDef("write", [], Prim("write"))
```

```
  ])
  => l
```

Counting

```
   AddCounter : SDef(f, xs, s) -> SDef(f, xs, Seq(s, CountRule(f)))
```

Specification to Definition List Desugaring a specification consist of deriving the list of joined strategy definitions from its rule definitions and strategy definitions; The signature components are ignored;

```
rules

  Sp0 : Specification(bspecs) -> bspecs
  Sp1 : Signature(bsigs)      -> <congdefs> bsigs
  Sp2 : Strategies(sdefs)     -> sdefs
  Sp3 : Rules(rdefs)          -> <map(RDtoSD)> rdefs

strategies

  spec-to-sdefs = Sp0;
                  map(Sp1 + Sp2 + Sp3);
                  AddPrimitives;
                  concat;
                  map(AddCounter);
                  map(strename)
```

3.5 Desugaring

Desugaring : translating high-level constructs to low-level ones

```
module desugar
imports sugar
imports stratlib
imports list-sort
```

Varyadic operators

```
rules

  HL : Seqs(Nil)             -> Id
  HL : Seqs(Cons(s, ss))     -> Seq(s, Seqs(ss))
  HL : Choices(Nil)          -> Fail
  HL : Choices(Cons(s, ss))  -> Choice(s, Choices(ss))
  HL : LChoices(Nil)         -> Fail
  HL : LChoices(Cons(s, ss)) -> LChoice(s, LChoices(ss))
```

```
  MkSeq : (s1, s2) -> Seq(s1, s2)

strategies

  seqs = foldr(build(Id), MkSeq)
```

Match, Build, Apply combinations

```
rules

  HL : BA(s, t)       -> Seq(Build(t), s)
  HL : MA(t, s)       -> Seq(Match(t), s)
  HL : AM(s, t)       -> Seq(s, Match(t))
  HL : BAM(s, t1, t2) -> Seqs([Build(t1), s, Match(t2)])
```

Strategy Applications Factoring out strategy applications; The right-hand side of a rule can contain applications of a strategy to a term; This is factored out by translating it to a condition that applies the strategy and matches the result against a new variable, which is then used in the rhs; In fact this can be generalized to applications in arbitrary builds;

```
rules

  Bapp0 : Build(t[App(Build(t'), t'')]) -> Build(t[t'])

  Bapp1 : Build(App(s, t')) -> Seq(Build(t'), s)

  Bapp2 : Build(t[App(s, t')]) ->
          Scope([x], Seq(BAM(s, t', Var(x)), Build(t[Var(x)])))
          where new => x
```

Rules to Strategies A rule corresponds to a strategy that first matches the left-hand side, then checks the conditions and finally builds the right-hand side; The left-hand side and right-hand side should be in basic term format, as defined by the predicate —bterm—;

```
strategies

  pureterm = not(topdown(Con(id, id, id) + App(id, id)))

  buildterm = not(topdown(Con(id, id, id) + Wld))

rules

  RtoS : SRule(Rule(l, r, s)) -> Seqs([Match(l), Where(s), Build(r)])
         where <pureterm> l ; <buildterm> r

  RtoS : SRule(StratRule(l, r, s)) -> Seqs([l, Where(s), r])
```

Contexts Factoring out contexts; Contexts used in a rule are translated to a local traversal that replaces the pattern occuring in the context in the lhs by the pattern occurring in the context in the rhs;

```
rules

  Rcon : SRule(Rule(l[Con(Var(c), l', f)], r[Con(Var(c), r', f')], s)) ->
         Scope([c'],SRule(
           Rule(l[Var(c)], r[Var(c')],
                Seq(s, BAM(Call(f', [SRule(Rule(l', r', Id))]),
                           Var(c), Var(c'))))))
         where new => c'

  Rcon' : SRule(Rule(l[Con(Var(c), l', f)], r[Con(Var(c), r', f')], s)) ->
         SRule(Rule(l[Var(c)],
                    r[App(Call(f', [SRule(Rule(l', r', Id))]), Var(c))],
                    s))

  Rcon'' : SRule(Rule(l[Con(Var(c), l', f)], r, s)) ->
         SRule(Rule(l[Var(c)], r,
               Seq(s, BA(Call(f, [Match(l')]), Var(c)))))
        (* Con(Var(c), _, _) should not occur in r *)
```

Note: The local traversal should be closed for variables not occuring in the outer pattern; But this is more relevant for multi-contexts which are not supported yet;

Other problems:

- local variables for inner rule; the inner SRule should be enclosed in a `Scope(xs, _)` where

```
   <diff> (<tvars> (l', r'), <tvars> (l[Var(c)], r[Var(c')])) => xs
```

- placement of derived strategy in where clause; option first do a matching traversal at start of where, and at end of where do a replacing traversal.

- multiple uses of context in rhs

Desugaring single rules

```
strategies

  desugarRule = rec x(try(Rcon; x + Scope(id, x) + RtoS))
```

Desugaring Strategies

```
strategies

  desugar = topdown(try(desugarRule); repeat(HL + (Bapp0 <+ Bapp1 <+ Bapp2)))
```

```
  desugar' = topdown(try(desugarRule); repeat(HL))

  desugar-spec = map(SDef(id, id, desugar))
```

3.6 Needed Definitions

Extract those definitions that are needed for the strategy .

```
module needed-defs
imports strategy
imports sugar
imports stratlib
imports list-set
imports list-misc

rules

  Init: defs -> (["main"], ["main"], [], defs)

  Next: (Cons(f, needed), seen, ndefs, defs) ->
        (needed'', seen', Cons(<joindefs> fdefs, ndefs), defs)
        where
          <filter(match(SDef(f,_,_)))> defs => fdefs;
          <diff> (<svars> fdefs, seen) => needed';
          <conc> (needed', needed) => needed'';
          <conc> (needed', seen) => seen'

  End : ([], seen, ndefs, defs) -> ndefs

  JoinDefs1 : [sdef] -> sdef

  JoinDefs2 : Cons(SDef(f, xs, s), defs) -> SDef(f, ys, Choices(ss))
              where <map(new)> xs => ys ;
                    <map(MkCall)> ys => ys' ;
                    <map({zs, s: <<SDef(f, zs, s) -> <ssubs> (zs, ys', s)>>})>
                     Cons(SDef(f, xs, s), defs) => ss

  MissingDef : (Cons(f, fs), seen, ndefs, defs) ->
               Error(["operator ", f, " undefined "])

strategies

  joindefs = JoinDefs1 <+ JoinDefs2

  needed-defs = Init; repeat(Next); try(End)
                <+ MissingDef;
                   FAIL
```

3.7 Inlining

In this module we define inlining of let defined strategy definitions;

```
module inlining
imports strategy
imports stratlib
imports lib

rules

  Inl1 : Let(SDef(f, [], s1), s2[Call(SVar(f), [])]) ->
         Let(SDef(f, [], s1), s2[<strename> s1])

  Inl2 : Let(SDef(f, Cons(x, xs), s1), s2[Call(SVar(f), ss)]) ->
         Let(SDef(f, Cons(x, xs), s1),
             s2[<ssubs ; strename> (Cons(x, xs), ss, s1)])

  Inl3 : Let(SDef(f, Cons(x, xs), s1), s2[Call(SVar(f), ss)]) ->
         Let(SDef(f, Cons(x, xs), s1),
             s2[<zip(MkSdef) ;
                 foldr(build(s1), MkSDef ; Inl1) ;
                 strename> (Cons(x, xs), ss)])

  MkSDef : (x, s) -> SDef(x, [], s)

  Dead : Let(SDef(f, xs, s1), s2) -> s2
         where <not(in)> (SVar(f), s2)

strategies

  inline2  = bottomup(repeat((repeat1(Inl1 <+ Inl2) ; try(Dead)) <+ Dead))

  inline2 = bottomup(repeat(repeat1(Inl1 + Inl2)))
```

Expand the strategy with respect to the desugared specification;

(*** Is this strategy correct?? ***)

```
rules

  InitExpand' : env -> (Call(SVar("main"), []), env)

  InitExpand : env -> (env1, env2)
               where <repeat(SplitDefs <+ SplitDefs')> (env, [], [])
                     => ([], env1, env2)

  SplitDefs : (Cons(sdef, sdefs), env1, env2) ->
              (sdefs, env1, Cons(sdef, env2))
              where <doInline> sdef
```

```
  SplitDefs' : (Cons(sdef, sdefs), env1, env2) ->
               (sdefs, Cons(sdef, env1), env2)

  ExpandCall : (Call(SVar(f), ss), env) ->
               (<strename> s, <ExtendEnv> (xs, ss, env))
               where <fetch(match(SDef(f, xs, s)))> env

  ExpandCall : (Call(SVar(f), ss), env) ->
               (Call(SVar(g), ss), env)
               where <fetch(match(SDef(f, [], Call(SVar(g), []))))> env

  TryCall : (Call(SVar(f), ss), env) -> (Tried(Call(SVar(f), ss)), env)

  Dist(s) : (e, env) -> (<all({x:<<x -> (x, env)>>} ; s ; Fst)> e, env)

  ExtendEnv : (xs, ss, env) -> <conc> (<zip(MkSDef)> (xs, ss), env)
```

Heuristics for inlining; inline all operators with arguments. Also nullary operators that represent rules (do a match as first action).

Todo: Also inline sums of rules if they occur inside a sum

```
strategies

  doInline = SDef(id, Cons(id, id), id) +
             SDef(not("main"), [], Scope(id, Seq(Match(id), id))) +
             SDef(not("main"), [], Seq(Scope(id, Seq(Match(id), id)), id))

  expandStrat = rec eval(Dist(eval); try(repeat1(ExpandCall); eval))

  expandStrat' = (rec eval(Dist(eval) ; ((repeat1(ExpandCall) ; eval)
                                          <+ TryCall <+ id))) ; Fst

  inline = InitExpand; expandStrat; Fst; rename_sdefs

rules

  rename_sdefs :
    sdefs -> sdefs'
    where <filter(NewName)> sdefs => tbl;
          <map((id, MkCall))> tbl => sbs;
          <map(RenameSDef(!tbl, !sbs))> sdefs => sdefs'

  NewName : SDef(x, _, _) -> (x, <new>())
            where not(!x => "main")

  RenameSDef(mktbl, mksbs) :
    SDef(x, xs, s) ->
    SDef(y, xs, <subs'(IsSVar, mksbs)> s)
    where (mktbl; fetch(?(x, y))) <+ (!x => y)
```

Chapter 4

Optimizer

4.1 Optimizer

```
module optimizer
imports simplification
imports optimization
imports optimization3
imports match-build

strategies

  main = (* defmb; *) optimize
```

4.2 Simplification

This module specifies basic simplification rules for strategies.

```
module simplification
imports strategy
```

Identity

```
rules
  I1 : Test(Id)         -> Id
  I2 : Not(Id)          -> Fail
  I3 : Seq(Id, s)       -> s
  I4 : Seq(s, Id)       -> s
  I5 : Choice(s, Id)    -> s
  I6 : Choice(Id, s)    -> s
  I7 : LChoice(Id, s)   -> Id
  I8 : Scope(xs, Id)    -> Id
  I9 : Rec(x,Id)        -> Id
  I10 : All(Id)         -> Id
```

```
  I11 : Path(i,Id)      -> Id
  I12 : Where(Id)       -> Id
  I13 : Cong(f, ss)     -> MatchFun(f) where <map(match(Id))> ss
  I14 : CongWld(ss)     -> Id where <map(match(Id))> ss
  I15 : App(Id, t)      -> t
  I16 : Match(Wld)      -> Id
strategies

  I = I1 + I2 + I3 + I4 + I5 + I6 + I7 + I8 + I9 + I10 +
      I11 + I12 + I13 + I14 + I15 + I16
```

Note that the following rules are not sound

```
rules
  NotValid : One(Id)          -> Id
  NotValid : Some(Id)         -> Id
```

Failure

```
rules
  F1  : Test(Fail)       -> Fail
  F2  : Not(Fail)        -> Id
  F3  : Seq(Fail, s)     -> Fail
  F4  : Seq(s, Fail)     -> Fail
  F5  : Scope(xs, Fail)  -> Fail
  F6  : Rec(x,Fail)      -> Fail
  F7  : Some(Fail)       -> Fail
  F8  : One(Fail)        -> Fail
  F9  : Path(i,Fail)     -> Fail
  F10 : Cong(f, ls)      -> Fail where <fetch(match(Fail))> ls
  F11 : Choice(Fail, s)  -> s
  F12 : Choice(s, Fail)  -> s
  F13 : LChoice(Fail, s) -> s
  F14 : LChoice(s, Fail) -> s
  F15 : Where(Fail)      -> Fail
  (* F : Case([])         -> Fail *)
strategies

  F = F1 + F2 + F3 + F4 + F5 + F6 + F7 + F8 + F9 + F10 +
      F11 + F12 + F13 + F14 + F15
```

Note that the following rule is not sound

```
rules

  NotValid : All(Fail) -> Fail
```

Commutativity and Associativity

```
rules

  Comm : Choice(x, y) -> Choice(y, x)

  Ass : Choice(Choice(x, y), z)   -> Choice(x, Choice(y, z))
  Ass : Seq(Seq(x, y), z)         -> Seq(x, Seq(y, z))
  Ass : LChoice(LChoice(x, y), z) -> LChoice(x, LChoice(y, z))

  LAss: Seq(s1, Seq(s2, s3)) -> Seq(Seq(s1, s2), s3)
```

Idempotence

```
  P : Choice(x, x)  -> x
  P : LChoice(x, x) -> x

  P' : Choice(s, s'[s]) -> s'[s](conChoice)
  P' : LChoice(s, s'[s]) -> LChoice(s, s'[Id](conLChoice))

  P : Where(Where(s))  -> Where(s)
  P : Not(Not(s)) -> Test(s)
  P : Test(Test(s)) -> Test(s)

  P : Where(Seq(Where(s1), Seq(Build(t), s2))) ->
      Where(Seq(s1, Seq(Build(t), s2)))
```

4.3 Distribution

```
module distribution
imports strategy
imports stratlib
```

Choice and Sequential Compostion The following rules are the usual formulations for distributivity.

```
rules

  D : Seq(x, Choice(y, z))  -> Choice(Seq(x, y), Seq(x, z))
  D : Seq(x, LChoice(y, z)) -> LChoice(Seq(x, y), Seq(x, z))

  D : LChoice(Choice(x, y), z) -> Choice(LChoice(x, LChoice(y, z)),
                                         LChoice(y, LChoice(x, z)))

  D : LChoice(x, Choice(y, z)) -> Choice(LChoice(x, y), LChoice(x, z))
```

The corresponding right-distributivity laws are not sound with respect to the semantics.

```
  Dn : Seq(Choice(x, y), z)      -> Choice(Seq(x, z), Seq(y, z))
  Dn : Seq(LChoice(x, y), z)     -> LChoice(Seq(x, z), Seq(y, z))
```

For optimization we want to apply these rules in the reverse direction, i.e., factor out common prefixes of two alternatives.

```
  D' : LChoice(Seq(s, s1), Seq(s, s2)) -> Seq(s, LChoice(s1, s2))

  D' : LChoice(Seq(s, s1), LChoice(Seq(s, s2), s3)) ->
       LChoice(Seq(s, LChoice(s1, s2)), s3)

  D' : Seq(LChoice(s1, s2), CountRule(x)) ->
       LChoice(Seq(s1, CountRule(x)), Seq(s2, CountRule(x)))

  ChoiceMergeXXX :
        Choice(Seq(s, s1), Seq(s, s2)) -> Seq(s, Choice(s1, s2))

  ChoiceMergeXXX :
        Choice(Seq(s, s1), Choice(Seq(s, s2), s3)) ->
        Choice(Seq(s, Choice(s1, s2)), s3)

  ChoiceMerge :
        Choice(Seq(s, s1), cs[Seq(s, s2)]) ->
        Choice(Seq(s, Choice(s1, s2)), cs[Fail](conCM))

strategies

  conCM(s) = rec x(s <+ {x: (Choice(s, ?x) + Choice(?x, s)); !x} <+
                   (Choice(x, id) + Choice(id, x)))
```

Term Traversal All and Some

```
rules

  T : Seq(All(x), All(y))      -> All(Seq(x,y))
  T : Choice(One(x), One(y))   -> One(Choice(x, y))
```

Path

```
  D' : Seq(Path(i, x), Path(i, y)) -> Path(i, Seq(x, y))

  C : Seq(Path(i, x), Path(j, y)) -> Seq(Path(j, y), Path(i, x))
      where not(<eq> (i, j))
```

Scope

Scope lifting

note: scopes cannot be lifted out of recs

```
  S': Scope(Nil, s) -> s
  S : Scope(xs, Scope(ys, s)) -> Scope(<conc> (xs, ys), s)
  S : Seq(Scope(xs, s1), Scope(ys, s2)) -> Scope(<conc> (xs, ys), Seq(s1, s2))
```

```
S' : Scope(xs, Seq(MatchFun(f), s2)) -> Seq(MatchFun(f), Scope(xs, s2))

S' : Scope(xs, s) -> Scope(ys, s)
    where <intersect> (xs, <tvars> s) => ys

S' : Scope(xs, Seq(Build(Var(x)), s)) ->
    Seq(Build(Var(x)), Scope(xs, s))
    where <not(in)> (x, xs)

S' : Scope(xs, Seq(Match(Var(x)), s)) ->
    Seq(Match(Var(x)), Scope(xs, s))
    where <not(in)> (x, xs)

S : Seq(Scope(xs, s), CountRule(n)) -> Scope(xs, Seq(s, CountRule(n)))

MkCase1 : Seq(MatchFun(f), s) -> [(f, s)]

MkCase2(x) : Choice(Seq(MatchFun(f), s), s') ->
             Cons((f, s), <x> s')

Cs : Choice(s1, s2) -> Case(cases)
     where <rec x(MkCase1 + MkCase2(x))> Choice(s1, s2) => cases

CaseMerge : Choice(Seq(MatchFun(f), s1), Seq(MatchFun(g), s2)) ->
            Case([(f, s1), (g, s2)])
            where <not(eq)> (f, g)

CaseMerge : Choice(Seq(MatchFun(f), s1), Case(cases[(f, s2)])) ->
            Case(cases[(f, Choice(s1, s2))](fetch))

CaseMerge : Choice(Case(cases[(f, s1)]), Seq(MatchFun(f), s2)) ->
            Case(cases[(f, Choice(s1, s2))](fetch))

CaseMerge'' : Choice(Case(cs1[(f, s1)]),
                     Case(cs2[Cons((f, s2), cs3)]))
            ->
            Choice(Case(cs1[(f, Choice(s1, s2))](fetch)),
                   Case(cs2[cs3](at_suffix)))

CaseMerge : Choice(Case(cs1), Case(cs2)) -> Choice(Case(cs1'), Case(cs2'))
            where
            <fetch({f, s1:
                    <<(f, s1) -> (f, Choice(s1, s2))
                      where <at_suffix({cs3: <<Cons((f, s2), cs3) -> cs3>>})> cs2 => cs2'
                    >>})
            > cs1 => cs1'
```

When there is no overlap between the guards of the two case expressions the cases can be merged.

```
CaseMerge' : Choice(Seq(MatchFun(f), s), Case(cases)) ->
             Case(Cons((f, s), cases))

CaseMerge' : Choice(Case(cases), Seq(MatchFun(f), s)) ->
             Case(Cons((f, s), cases))

CaseMerge' : Choice(Case(cases1), Case(cases2)) ->
             Case(<conc> (cases1, cases2))
```

Left choice of cases

```
CaseMerge : LChoice(Seq(MatchFun(f), s1), Case(cases[(f, s2)])) ->
            Case(cases[(f, LChoice(s1, s2))](fetch))

CaseMerge : LChoice(Case(cases[(f, s1)]), Seq(MatchFun(f), s2)) ->
            Case(cases[(f, LChoice(s1, s2))](fetch))

CaseMerge'' : LChoice(Case(cs1[(f, s1)]),
                      Case(cs2[Cons((f, s2), cs3)]))
            ->
            LChoice(Case(cs1[(f, LChoice(s1, s2))](fetch)),
                    Case(cs2[cs3](at_suffix)))

CaseMerge : LChoice(Case(cs1), Case(cs2)) -> LChoice(Case(cs1'), Case(cs2'))
            where
            <fetch({f, s1:
                    <<(f, s1) -> (f, LChoice(s1, s2))
                      where <at_suffix({cs3: <<Cons((f, s2), cs3) -> cs3>>})> cs2 => cs2'
                    >>})
            > cs1 => cs1'

CaseMerge' : LChoice(Seq(MatchFun(f), s), Case(cases)) ->
             Case(Cons((f, s), cases))

CaseMerge' : LChoice(Case(cases), Seq(MatchFun(f), s)) ->
             Case(Cons((f, s), cases))

CaseMerge' : LChoice(Case(cases1), Case(cases2)) ->
             Case(<conc> (cases1, cases2))
```

Recursion

UnFolding

```
U : Rec(x,s) -> <subs> (x, Rec(x,s), s)
```

Folding (This does probably not work)

```
Fold : s[Rec(x, s')] -> s'
       where build(Rec(x, s[Call(x,[])])) => s'
```

```
        (* Change x, SVar(x) to Label(SVar(x), [])) *)

  D'' : Seq(Rec(x, LChoice(Seq(s1, SVar(x)), s2)), s3) ->
       Rec(x, LChoice(Seq(s1, SVar(x)), Seq(s2, s3)))
          where <not(in)> (SVar(x), s2)

  D'' : Seq(Rec(x, LChoice(s1, Seq(s2, SVar(x)))), s3) ->
       Rec(x, LChoice(Seq(s1, s3), Seq(s2, SVar(x))))
       where <not(in)> (SVar(x), s2)

  D'' : Seq(Rec(x, Choice(Seq(s1, Label(SVar(x), [])), s2)), s3) ->
       Rec(x, LChoice(Seq(s1, x), Seq(s2, s3)))
       where <not(in)> (x, s2)

  D'' : Seq(Rec(x, Choice(s1, Seq(s2, x))), s3) ->
       Rec(x, Choice(Seq(s1, s3), Seq(s2, x)))
       where <not(in)> (x, s2)
```

4.4 optimization3

```
module optimization3
imports strategy

rules

(* Decomposing Match *)

  MkVar : x -> Var(x)

  M : Match(Op(f, ts)) -> Seq(Match(Op(f, xs)), ms)
      where <map(new ; MkVar)> ts => xs ;
            <zip(M')> (xs, ts) => ms

  M' : (x, t) -> Seq(Build(x), Match(t))

(* Merging Matches *)

  M : Choice(Seq(Match(Op(f, xs)), s1),
             Seq(Match(Op(f, xs)), s2)) ->
      Seq(Match(Op(f, xs)), Choice(s1, s2))
```

4.5 Optimization

```
module optimization
imports strategy
```

```
imports list
imports simplification
imports distribution
imports stratlib
```

match and build

```
rules

  M : Cong(f, ss) -> Match(Op(f, ts))
      where <map({t:<<Match(t) -> t>>} + <<Id -> Match(Wld)>>)> ss => ts

  M : Seq(Build(t), Seq(Prim("new"), s)) -> Seq(Prim("new"), s)

  M : Seq(Match(t), s[Build(t)]) -> Seq(Match(t), s[Id](firstInSeq))
  M : Seq(Build(t), s[Match(t)]) -> Seq(Build(t), s[Id](firstInSeq))

  M : Seq(Build(t), s[Build(t')]) -> s[Build(t')](firstInSeq)

  M : Seq(Build(Op(f, ts)), s[MatchFun(f)]) ->
      Seq(Build(Op(f, ts)), s[Id](firstInSeq))

  M' : App(s[MatchFun(f)], Op(f, ts)) ->
      App(s[Id](firstInSeq), Op(f, ts))

  M : Seq(Match(Var(x)), s[Build(Var(x))]) ->
      Seq(Match(Var(x)), s[Id](firstInSeq))

(*
  M' : Seq(Build(t1), Match(t2)) -> Fail
       where <not(match(t1))> t2
*)

  M' :
    Scope(xs, Seq(Build(Var(y)), Seq(Match(Var(x)), s))) ->
    Scope(<diff> (xs, [x]),
          Seq(Build(Var(y)), <tsubs> ([Var(x)], [Var(y)], s)))
    where <in> (x, xs)

  CommonSubterm :
    Seq(Match(t1[Op(f, ts)]), s[Build(t2[Op(f, ts)])]) ->
    Scope([x],
      Seq(Match(t1[Var(x)]),
      Seq(Seq(Build(Var(x)), Seq(Match(Op(f, ts)),
          s[Build(t2[Var(x)])])))))
    where new => x

(* As(x, t) : match(t) and bind to x *)

  CommonSubterm :
```

```
  Seq(Match(t1[Op(f, ts)]), s[Build(t2[Op(f, ts)])]) ->
  Scope([x],
    Seq(Match(t1[As(Var(x), Op(f, ts))]),
        s[Build(t2[Var(x)])]))
  where new => x
```

Match-Build Fusion (This did not quite work as portrayed here. The problem is that the congruence may have a side-effect that influences the code in the build. The MkApp rule creates an application only if all variables are defined in the branch.)

```
MkApp : (s, t) -> Seq(s, Build(t))
        where <diff> (<tvars> t, <tvars> s) => []

CongBuild :
      Seq(Cong(f, ss), s[Build(Op(f, ts))]) ->
      Seq(Cong(f, <zip(MkApp)> (ss, ts)), s[Id](firstInSeq))

CongBuild :
      Seq(MatchFun(f), Seq(CongWld(ss), s[Build(Op(f, ts))])) ->
      Seq(MatchFun(f), Seq(CongWld(<zip(MkApp)> (ss, ts)), s[Id](firstInSeq)))
```

Build-Congruence fusion

```
BuildCong :
      Seq(Build(Op(f, ts)), s[Cong(f, ss)]) ->
      Seq(Build(Op(f, <zip({s, t:<<(s, t) -> App(s, t)>>})> (ss, ts))),
          s[Id](firstInSeq))

BuildCong :
      Seq(Build(Op(f, ts)), s[CongWld(ss)]) ->
      Seq(Build(Op(f, <zip({s, t:<<(s, t) -> App(s, t)>>})> (ss, ts))),
          s[Id](firstInSeq))

BuildCong :
      Seq(Build(Op(f, ts)), Seq(MatchFun(f), s[CongWld(ss)])) ->
      Seq(Build(Op(f, <zip({s, t:<<(s, t) -> App(s, t)>>})> (ss, ts))),
          s[Id](firstInSeq))

AppCong :
      App(s[Cong(f, ss)], Op(f, ts)) ->
      App(s[Id](firstInSeq),
          Op(f, <zip({s, t:<<(s, t) -> App(s, t)>>})> (ss, ts)))

AppCong :
      App(s[CongWld(ss)], Op(f, ts)) ->
      App(s[Id](firstInSeq),
```

```
          Op(f, <zip({s, t:<<(s, t) -> App(s, t)>>})> (ss, ts)))

  AppCong :
        App(Seq(MatchFun(f), s[CongWld(ss)]), Op(f, ts)) ->
        App(s[Id](firstInSeq),
            Op(f, <zip({s, t:<<(s, t) -> App(s, t)>>})> (ss, ts)))
```

Strategies for application of simplification rules.

```
strategies

  simplify = I + F + Ass + P + S + D' + ChoiceMerge (* + M *)

  fuse = I + Ass + CongBuild + BuildCong + AppCong

  optimize' = downup(repeat(simplify));
              downup2(try(Cs), repeat(CaseMerge))

  optimize' = downup(repeat(fuse));
              downup(repeat(simplify + (CaseMerge <+ CaseMerge')))

  optimize' = downup(repeat(fuse))

  optimize' = downup(repeat(simplify + (CaseMerge <+ CaseMerge')))

  optimize = downup(repeat(simplify))

  optimize' = downup(repeat(I + F (* + Ass + P + S + D' + ChoiceMerge *)))

  optimize' = downup(repeat(I12 + I13 + F15))
```

4.6 matching-tree

```
module matching-tree
imports strategy stratlib optimization instructions fixpoint-traversal
signature
  operations
    Case : List(Strat) * Strat -> Strat
    PreCase : Strat * Strat -> Strat
    Double : a * a -> a
```

Join branches of a choice that start with a variable match. Make the single joined branch starting with a variable match the first branch in the choice.

```
strategies

  MatchVarPrefix'(s) =
        rec x(Seq(Match(Var(id)), x) <+ (Ass + I); x <+
              {s': ?s'; !Choice(s', <s>())})
```

```
  MatchVarPrefix(s) =
        rec x(Seq(Match(Var(id)), x) <+ (Ass + I); x <+
              {s': ?s'; s; MatchVarPrefix'(!s')})

  ChoiceMerge' = ChoiceMerge1 + ChoiceMerge2 + ChoiceMerge3

 (* Note: better if this can be + *)

rules

  ChoiceMerge1 :
        Choice(Seq(Match(Var(x)), s1), cs) ->
        Choice(Seq(Match(Var(x)), Seq(Match(Var(y)), s1')), cs')
        where <conChoice(?Seq(Match(Var(y)), s2); !Fail)> cs => cs';
              <MatchVarPrefix(!s2)> s1 => s1'

  ChoiceMerge2 :
        Choice(Seq(Match(Var(x)), s1), cs) ->
        Choice(Seq(Match(Var(x)), s1'), cs')
        where <conChoice(?Seq(Match(Wld), s2); !Fail)> cs => cs';
              <MatchVarPrefix(!s2)> s1 => s1'

  ChoiceMerge3 :
        Choice(Seq(Match(t), s2), cs) ->
        Choice(Seq(Match(Var(x)), s1), cs')
        where not(!t => Var(_));
              <conChoice(?Seq(Match(Var(x)), s1); !Seq(Match(t), s2))> cs => cs'
```

PreCase Introduction Assuming that the branches of the choice have been merged as described above, the choice is turned into a precase. The first branch of the precase deals with the cases where the first match is to a non-variable term. If the outermost function symbol of the subject term matches any of the outermost function symbols of the branches, one of these branches will be taken. The second branch of the precase deals with those subject terms that have an outermost function symbol that matches non of the patterns.

```
rules

  AddVarPart(bld_s) :
        Seq(Match(t), s) -> Seq(Match(t), Choice(s, <bld_s> ()))

  AddVarPart'(bld_s) :
        x -> (x, <bld_s> ())

  SepVars :
        Choice(Seq(Match(Var(x)), s), cs) ->
        Seq(Match(Var(x)), Seq(s1, PreCase(cs', s2)))
        where <rec x(Seq(Match(Var(id)), x) <+ (Ass + I); x
```

```
                <+ <<s2 -> Id>>)> s => s1;
            <rec x(AddVarPart(build(s2)) <+
                Choice(x, x) <+
                AddVarPart'(build(s2)))> cs => cs'

  SepVars :
      Seq(Match(Var(x)), s) -> Seq(Match(Var(x)), s)

  SepVars' : cs -> PreCase(cs, Fail)

  PreCaseSimp : PreCase(PreCase(cs, s), s') -> PreCase(cs, LChoice(s, s'))

  PreCaseSimp : PreCase(Seq(s1, s2), Fail) -> Seq(s1, s2)

  (* PreCaseSimp : PreCase(Id, s) -> Fail *)

  Duplicate(s) : x -> Double(x, <s> x)
```

Definining Operator Matching Breaking up a match of an application into a match of the function symbol and the argument terms. Stack operations are used to traverse the subject term.

```
rules

  MatchOp :
      Match(Op(f, ts)) ->
      Seq(MatchFun(f), <rec x(MatchOpAux(x))> (ts, 0))

  MatchOp :
      AnnMatch(t1, t2) ->
      Seq(Tpush, Seq(Build(t1), Seq(GetAnn, Seq(Match(t2), Tpop))))

  MatchOpAux(x) :
      (Nil, n) -> Id
  MatchOpAux(x) :
      (Cons(t, ts), n) ->
      Seq(Arg(n), Seq(Match(t), Seq(Tpop, <x> (ts, <plus> (n, 1)))))
```

PreCase to Case A precase has a choice as first branch. This is converted to a list of branches in a Case. A Case strategy chooses one of the branches in the first argument based on the first action in those branches. This first choice is binding, i.e., after a branch is chosen, no backtracking to other branches is done.

```
rules

  MkCaseA : x -> [x]
            (* where !x; Seq(MatchFun(id) + Match(Int(id) + Str(id)), id) *)
```

```
  MkCaseB(x) : Choice(s1, s2) -> <conc> (<x> s1, <x> s2)

  MkCase : PreCase(s1, s2) -> Case(cases, s2)
           where <rec x(MkCaseB(x) <+ MkCaseA)> s1 => cases
```

One Level

```
strategies

  is-case = Choice(rec x(Seq(Match(id), id) + Choice(x, x)),
                   rec x(Seq(Match(id), id) + Choice(x, x)))

  match-tree1 =
        is-case;
        repeat(ChoiceMerge');
        (* topdown(repeat(ChoiceMerge' + F)); *)
        (* downup(repeat(ChoiceMerge' + F + Ass + I)); *)
        (* try choicetd or other choice specific traversal *)
        (* choicebu-l(repeat(ChoiceMerge' + F)); *)
        (* downup(repeat(ChoiceMerge'; pseudo-innermost3(simplify))); *)
        pseudo-innermost3(simplify);
        (SepVars <+ Choice(id, id); SepVars');
        rec z(try(Seq(id, z) <+ PreCase(z, id) <+ Id <+ Fail <+
              choicemap(Seq(MatchOp +
                            Match(Wld + Var(id) + Int(id) + Str(id)), id))))

  match-tree' =
        repeat(oncetd(match-tree1);
               innermost'(simplify + PreCaseSimp));
        topdown(try(MkCase))

  match-tree' =
        topdown(try(match-tree1;
                    innermost'(simplify + PreCaseSimp)));
        topdown(try(MkCase))

  match-tree =
        topdown(try(match-tree1;
                    pseudo-innermost3(simplify + PreCaseSimp)));
        topdown(try(MkCase))

  match-tree' =
        alltd(match-tree1)
```

All Levels

```
rules
```

```
  LiftScope : Choice(Scope(xs, s1), Scope(ys, s2)) ->
              Scope(<conc> (xs, ys), Choice(s1, s2))

  IntroContinuation :
        Seq(Match(t), s) ->
        Let(SDef(x, [], s), Seq(Match(t), Call(SVar(x), [])))
        where new => x

  LiftLet : Choice(Let(sdef, s), s') -> Let(sdef, Choice(s, s'))
  LiftLet : Choice(s', Let(sdef, s)) -> Let(sdef, Choice(s', s))

        (* assuming sdef does not affect s' *)

  JoinVars :
        Scope(xs, s[Seq(Match(Var(x)), Seq(Match(Var(y)), s'))]) ->
        Scope(<diff> (xs, [y]),
              <tsubs> ([y], [Var(x)], s[Seq(Match(Var(x)), s')]))

  MakeLinear :
        Scope(xs, Seq(Match(t), s)) ->
        Scope(Cons(y, xs), Seq(Match(t'),
                               Seq(Build(Var(y)), Seq(Match(Var(x)), s))))
        where
        <oncetd(Op(id, at_suffix(Cons(oncetd(?Var(x)),
                       oncetd(?Var(x); Var(<new => y>()))))))>
        t => t'

strategies

  is-scoped-case =
        Choice(rec x(Scope(id, Seq(Match(id), id)) + Choice(x, x)),
               rec x(Scope(id, Seq(Match(id), id)) + Choice(x, x)))

  lift-scopes =
        choicebu(LiftScope)

  lift-continuations =
        choicemap(IntroContinuation);
        rec x(bottomup(try(LiftLet; all(x))))

  make-linear =
        choicemap(repeat(MakeLinear))

  mk-match-tree =
        is-scoped-case;
        make-linear;
        lift-scopes;
        Scope(id, lift-continuations;
                  rec x(Let(id, x) <+ match-tree));
```

```
      repeat(JoinVars);
      id

main = topdown(try(mk-match-tree <+ MatchOp))

main' =
      is-case;
      repeat(ChoiceMerge')
```

Chapter 5

Backend

5.1 Abstract Machine

In this section we define the abstract machine instructions and a simplifier and peephole optimizer for abstract machine programs.

5.1.1 Instructions

```
module instructions
imports list-cons
signature
  sorts Instr
  operations
    Label               : String -> Instr
    Goto                : String -> Instr
    Block               : List(Instr) -> Instr

    Iprim               : String -> Instr
    Iprim2              : String * String -> Instr
    ICountRule          : String -> Instr
```

Term stack

```
    Tpush               : Instr
    Tpop                : Instr
    Tdupl               : Instr (* duplicate top of T stack *)
    Tduplinv            : Instr (* duplicate top of T stack *)
    Tdrop               : Instr

    TpushIthSon         : Int -> Instr
    TputIthSon          : Int -> Instr
    IsAppl              : Instr
```

```
MatchVar            : Int -> Instr
MatchFun            : String -> Instr
MatchFunFC          : String * String -> Instr
BuildVar            : Int -> Instr
BuildFun            : String * Int -> Instr

GetAnn              : Instr
SetAnn              : Instr
RemoveAnn           : Instr
```

Choice stack

```
Cpush               : String -> Instr
Cpop                : Instr
Crestore            : Instr
Cjump               : Instr
```

Return stack

```
Rpush               : String -> Instr
Return              : Instr
```

Environment stack

```
Epush               : Int -> Instr
Epop                : Int -> Instr
```

Counters

```
NewCounter          : Instr
CloseCounter        : Instr
CounterOK           : Instr
```

Traversal

```
AllInit             : Instr
AllNextSon          : String -> Instr
AllBuild            : Instr

OneInit             : Instr
OneNextSon          : Instr
OneBuild            : Instr

SomeInit            : Instr
SomeNextSon         : String -> Instr
SomeBuild           : Instr

TravInit            : Instr
MatchTravInit       : Instr
MatchTravEnd        : Instr
OneMatchNextSon     : Instr
```

Maybe needed in future ?

```
    ProcCall    : String * List(Int) -> Instr
    DBlock      : List(RegDecl) * List(Instr) -> Instr
```

5.1.2 Instruction Simplification

```
module ins-simplification
imports instructions list
```

Block Flattening The following rules specify the flattening of nested blocks into a single block with a list of instructions. This transformation enables peephole optimization, which optimizes adjacent instructions.

The strategy `flatten-blocks` is designed to minimize stack-depth.

```
rules

  BlkFlat' : Block(is) -> Block(is')
             where <rec x(Nil + Cons(id, x) ; try(BlkFlatAux))> is => is'

  BlkFlatAux : Cons(Block(is1), is2) -> <conc> (is1, is2)

  BlkFlat1 : Cons(Block([]), is) -> is
  BlkFlat2 : Cons(Block(Cons(i, is)), is') ->
              Cons(i, Cons(Block(is), is'))

strategies

  flatten-blocks =
        rec x(try(Nil + Block(x) +
                  Cons(id, x);
                  rec y(try(BlkFlat1 + BlkFlat2; Cons(id, y); y))))
```

Peephole Optimization Some sequences of adjacent instructions can be reduced to an equivalent, but shorter (code size reduction) or more efficient sequence of instructions.

```
rules
  PH : Cons(MatchVar(n), Cons(BuildVar(n), is)) ->
       Cons(MatchVar(n), is)

  PH : Cons(MatchFun(f), Cons(TravInit, Cons(AllBuild, is))) ->
       Cons(MatchFun(f), is)
```

Environment stack operations

```
  PH : Cons(Epush(0), is) -> is
```

```
PH : Cons(Epop(0), is) -> is

PH : Cons(Epush(n), Cons(Epush(m), is)) ->
     Cons(Epush(<plus> (n, m)), is)

PH : Cons(Epop(n), Cons(Epop(m), is)) ->
     Cons(Epop(<plus> (n, m)), is)
```

Term stack operations

```
PH : Cons(Arg(n), Cons(Tpop, is)) -> is
```

Redundant jumps

```
PH : Cons(Goto(l), Cons(Label(l), is)) ->
     Cons(Label(l), is)
```

Merging Labels Two adjacent label definitions can be merged into a single label definitions. This requires substituting the label that is left for the label that is removed in all jumps to that label.

```
rules

  DblLbls : Cons(Label(l1), Cons(Label(l2), is)) ->
            Cons(LabelSubsU(l2, l1),
            Cons(Label(l1),
            Cons(LabelSubsD(l2, l1), is)))
```

Applying upward substitutions

```
  SbsU :   Cons(LabelSubsU(l2, l1), Cons(Label(l2), is)) ->
           Cons(LabelSubsU(l2, l1), Cons(Label(l1), is))

  SbsU :   Cons(LabelSubsU(l2, l1), Cons(Goto(l2), is)) ->
           Cons(LabelSubsU(l2, l1), Cons(Goto(l1), is))

  SbsU :   Cons(LabelSubsU(l2, l1), Cons(Cpush(l2), is)) ->
           Cons(LabelSubsU(l2, l1), Cons(Cpush(l1), is))

  SbsU :   Cons(LabelSubsU(l2, l1), Cons(Rpush(l2), is)) ->
           Cons(LabelSubsU(l2, l1), Cons(Rpush(l1), is))
```

Applying downward substitutions

```
  SbsD :   Cons(LabelSubsD(l2, l1), Cons(Label(l2), is)) ->
           Cons(Label(l1), Cons(LabelSubsD(l2, l1), is))

  SbsD :   Cons(LabelSubsD(l2, l1), Cons(Goto(l2), is)) ->
```

```
          Cons(Goto(l1), Cons(LabelSubsD(l2, l1), is))

  SbsD :  Cons(LabelSubsD(l2, l1), Cons(Cpush(l2), is)) ->
          Cons(Cpush(l1), Cons(LabelSubsD(l2, l1), is))

  SbsD :  Cons(LabelSubsD(l2, l1), Cons(Rpush(l2), is)) ->
          Cons(Rpush(l1), Cons(LabelSubsD(l2, l1), is))
```

Propagating substitutions up and down the list.

```
  Up :    Cons(i, Cons(LabelSubsU(l1, l2), is)) ->
          Cons(LabelSubsU(l1, l2), Cons(i, is))
          where not (<id> i => LabelSubsU(_,_))

  Down :  Cons(LabelSubsD(l2, l1), Cons(i, is)) ->
          Cons(i, Cons(LabelSubsD(l1, l2), is))
          where not (<id> i => LabelSubsD(_,_))
```

Removing substitutions at top and bottom of the list.

```
  UnSbsU : Cons(LabelSubsU(l2, l1), is) -> is

  UnSbsD : Cons(LabelSubsD(l2, l1), Nil) -> Nil
```

The Strategy The peephole optimization strategy.

```
strategies

  ph = rec y(try(PH; y + SbsU + (Up; (SbsU <+ Cons(id, y)))))

  phdown = repeat(PH + DblLbls + (SbsD <+ (UnSbsD + Down)))
  phup   = repeat(PH + (SbsU <+ rec x(Up; Cons(id, try(x)))))

  peephole' = Block(listbu(ph);
                    repeat(UnSbsU);
                    listtd(try(SbsD <+ (UnSbsD <+ Down))))

  peephole' = Block(listdu2(phdown, phup);
                    repeat(UnSbsU))

  peephole = try(Block(listdu(repeat(PH))))
```

5.2 Compilation

In this section we define the translation from strategies to abstract machine instructions.

(*** Consider transformation to an A-Normal-form like format. ***)

5.2.1 Backend

```
module backend
imports compiler specialized ins-simplification
strategies

  main = compile
```

Assembling the compiler. Translate Instr applications until done and flatten the blocks at the same time.

```
strategies

  (* compile : Strategy -> Instr *)

  compile = map(MkInstr;
                topdown(repeat(Cs <+ C)));
            Assemble;
            flatten-blocks;
            peephole
```

Make a strategy into an instruction

```
rules

  MkInstr : s -> Instr(s, [], [])

  Assemble : is ->
             Block([Rpush(ready),
                    Goto("main"),
                    Block(is),
                    Label(ready)])
             where new => ready
```

5.2.2 Compiler

```
module compiler
imports instructions strategy list subs
```

semi-instructions: strategies embedded into instructions

```
signature
  operations
    Instr : Strategy * Env * REnv -> Instr
```

Semi-instructions can be refined to instructions by means of the following rules.

Sequential Programming Identity and failure

```
rules

  C : Instr(Id, env, rcs) -> Block([])

  C : Instr(Fail, env, rcs) -> Goto("fail")
```

Test and negation

Note: this is really a where implementation the environment is not restored.

```
  C : Instr(Test(s), env, rcs) ->
      Block([Tdupl,
             Instr(s, env, rcs),
             Tpop])

  C : Instr(Not(s), env, rcs) ->
      Block([Cpush(fc),
             Tdupl,
             Instr(s, env, rcs),
             Cpop, Crestore, Cjump,
             Label(fc)])
      where new => fc
```

Sequential composition

```
  C : Instr(Seq(s1, s2), env, rcs) ->
      Block([Instr(s1, env, rcs),
             Instr(s2, env, rcs)])
```

Choice

```
  C : Instr(LChoice(s1, s2), env, rcs) ->
      Block([Cpush(fc),
             Instr(s1, env, rcs), Cpop, Goto(sc),
             Label(fc), Instr(s2, env, rcs),
             Label(sc)])
      where (new => sc) ; (new => fc)

  C : Instr(Choice(s1, s2), env, rcs) ->
      Block([Cpush(fc),
             Instr(s1, env, rcs), Cpop, Goto(sc),
             Label(fc), Instr(s2, env, rcs),
             Label(sc)])
      where (new => sc) ; (new => fc)
```

Cases (New Style)

```
  C : Instr(Case(cases, s), env, rcs) ->
      Block([Instr(Cases(cases, sc), env, rcs),
             Instr(s, env, rcs),
```

```
            Label(sc)])
    where new => sc

C : Instr(Cases([], sc), env, rcs) ->
    Block([])

C : Instr(Cases(Cons(Seq(s1, s2), cases), sc), env, rcs) ->
    Block([<CasePrefix> (s1, fc),
           Instr(s2, env, rcs),
           Goto(sc),
           Label(fc),
           Instr(Cases(cases, sc), env, rcs)])
    where new => fc

CasePrefix : (MatchFun(f),   fc) -> MatchFunFC(f, fc)
CasePrefix : (Match(Int(n)), fc) -> MatchIntFC(n, fc)
CasePrefix : (Match(Str(x)), fc) -> MatchStringFC(x, fc)
```

Cases (Old Style)

```
C' : Instr(Case(cases), env, rcs) ->
    Instr(Cases(cases, sc), env, rcs)
    where new => sc

C' : Instr(Cases([], sc), env, rcs) ->
    Label(sc)

C' : Instr(Cases([(f, s)], sc), env, rcs) ->
    Block([MatchFun(f),
           Instr(s, env, rcs),
           Label(sc)])

C' : Instr(Cases(Cons((f, s), cases), sc), env, rcs) ->
    Block([MatchFunFC(f, fc),
           Instr(s, env, rcs),
           Goto(sc),
           Label(fc),
           Instr(Cases(cases, sc), env, rcs)])
    where new => fc
```

Jumping

```
C : Instr(Rec(x, s), env, rcs) ->
    Block([Rpush(sc),
           Label(x),
           Instr(s, env, rcs),
           Return,
           Label(sc)])
    where new => sc
```

```
(*
  C : Instr(Call(SVar(x), []), env, rcs) ->
      Block([Rpush(ret), Goto(entry), Label(ret)])
      where <lookup> (x, rcs) => entry; new => ret
*)
  C : Instr(Call(SVar(x), []), env, rcs) ->
      Block([Rpush(ret), Goto(x), Label(ret)])
      where new => ret

  C : Instr(Let(sdef, s), env, rcs) ->
      Block([Instr(s, env, rcs),
             Goto(y),
             Instr(sdef, env, rcs),
             Label(y)])
      where new => y
```

This is not optimal! Causes a chain of jumps to the end of the code

```
  C : Instr(SDef(x, [], s), env, rcs) ->
      Block([Label(x),
             Instr(s, env, rcs),
             Return])
```

Path

```
  C : Instr(Path(i, s), env, rcs) ->
      Block([TpushIthSon(i),
             Instr(s, env, rcs),
             TputIthSon(i)])
```

Congruence

```
  C : Instr(Cong(f, ss), env, rcs) ->
      Block([MatchFun(f),
             TravInit,
             Instr(CongKids(ss), env, rcs),
             AllBuild])

  C : Instr(CongWld(ss), env, rcs) ->
      Block([TravInit,
             Instr(CongKids(ss), env, rcs),
             AllBuild])

  C : Instr(CongKids(Nil), env, rcs) ->
      Block([])

  C : Instr(CongKids(Cons(s, ss)), env, rcs) ->
      Block([OneNextSon,
             Instr(s, env, rcs),
             Instr(CongKids(ss), env, rcs)])
```

Generic traversal operators; All

```
C : Instr(All(s), env, rcs) ->
    Block([AllInit,
           Label(c1),
           AllNextSon(c2),     (* Succeed if there are no more children *)
           Instr(s, env, rcs),
           Goto(c1),           (* process next child *)
           Label(c2),
           AllBuild])
    where new => c1 ; new => c2
```

One

```
C : Instr(One(s), env, rcs) ->
    Block([IsAppl,
           OneInit,
           Label(c1),
           OneNextSon,        (* Fail if there are no more children *)
           Cpush(c1),
           Instr(s, env, rcs),
           Cpop,
           OneBuild])
    where new => c1 ; new => c2
```

Some

```
C : Instr(Some(s), env, rcs) ->
    Block([IsAppl,
           SomeInit,
           Label(c1),
           SomeNextSon(c2), (* jump to c2 if all children have been handled
                               and CounterOk *)
           Cpush(c1),
           Instr(s, env, rcs),
           Cpop,
           CounterOK, (* record success of at least one child *)
           Goto(c1),
           Label(c2),
           SomeBuild])
    where new => c1 ; new => c2
```

Scope

```
C : Instr(Scope(xs, s), env, rcs) ->
    Block([Epush(n),
           Instr(s, <conc> (xs, env), rcs),
           Epop(n)])
    where <length> xs => n
```

Where

```
C : Instr(Where(s), env, rcs) ->
    Block([Tdupl,
           Instr(s, env, rcs),
           Tpop])
```

Primitives

```
C : Instr(Prim(x), env, rcs) ->
    Iprim(x)

C : Instr(Prim2(x, y), env, rcs) ->
    Iprim2(x, y)

C : Instr(CountRule(x), env, rcs) ->
    ICountRule(x)
```

Matching terms

```
C : Instr(Match(Var(x)), env, rcs) ->
    MatchVar(n)
    where <get_index> (x, env) => n

C : Instr(MatchFun(f), env, rcs) ->
    MatchFun(f)

C : Instr(Match(Str(x)), env, rcs) ->
    MatchString(x)

C : Instr(Match(Int(n)), env, rcs) ->
    MatchInt(n)
```

Building Terms

```
C : Instr(Build(Str(x)), env, rcs) ->
    BuildStr(x)

C : Instr(Build(Int(x)), env, rcs) ->
    BuildInt(x)

C : Instr(Build(Var(x)), env, rcs) ->
    BuildVar(n)
    where <get_index> (x, env) => n

C : Instr(Build(Op(f, ts)), env, rcs) ->
    Block([BuildKids(ts, env, rcs),
           BuildFun(f, <length> ts)])

C : BuildKids(Nil, env, rcs) -> Block([])
```

```
C : BuildKids(Cons(t, ts), env, rcs) ->
    Block([Instr(Build(t), env, rcs), Tpush,
           BuildKids(ts, env, rcs)])

C : Instr(Build(App(s, t)), env, rcs) ->
    Block([Instr(Build(t), env, rcs),
           Instr(s, env, rcs)])
```

Term stack instructions

```
C : Instr(Tpop,   env, rcs) -> Tpop
C : Instr(Tpush,  env, rcs) -> Tpush
C : Instr(Arg(n), env, rcs) -> Arg(n)
```

Annotations

```
C : Instr(GetAnn, env, rcs) -> GetAnn

C : Instr(AnnBuild(t1, t2), env, rcs) ->
    Block([Tpush,
           Instr(Build(t1), env, rcs),
           Tpush,
           Instr(Build(t2), env, rcs),
           SetAnn])

C : Instr(AnnRemove(t), env, rcs) ->
    Block([Tpush,
           Instr(Build(t), env, rcs),
           RemoveAnn])
```

5.2.3 Special Patterns

This module defines exceptions to the normal compiler rules by recognizing patterns that can be implemented in a more efficient way.

```
module specialized
imports compiler simplification
```

Repeat A repetition of a strategy `s1` for 0 or more times, terminated with an application of a strategy `s2` can be translated into a loop structure. Only one `Cpush` is performed to deal with failure of the loop body. This has the effect that two copies of the subject term are on the stack. The loop body works on the top-most. If the body succeeds, this term can be committed and 'saved' by copying it over the term under the top of the stack.

The pattern `Repeat(s1, s2)` is an overlay that is defined below.

```
rules
```

```
Cs :: Instr(Repeat(?s1, ?s2), ?env, ?rcs) -->
      ! Block([Cpush(end_loop),
               Label(loop),
               Instr(s1, env, rcs),
               Tduplinv,
               Goto(loop),
               Label(end_loop),
               Instr(s2, env, rcs)])
      where new => loop; new => end_loop
```

This is an instance of tail recursion that can be further generalized to arbitrary many loop bodies. In that generalization we might consider the following rule which is not applicable in general because of backtracking:

```
Choice(Seq(s1, Call(SVar(x), [])), Seq(s2, Call(SVar(x), []))) ->
Seq(Choice(s1, s2), Call(SVar(x), []))
```

It is applicable when the strategies `s1` and `s2` are mutually exclusive, i.e., if `s1` succeeds, `s2` cannot possibly succeed.

The `Repeat(s1, s2)` overlay. The overlay recognizes the pattern of the `repeat` strategy operator from the library:

```
repeat(s) = rec x((s; x) <+ id)
```

but generalized to arbitrary terminators (other than `id`). The overlay transforms the first argument of the `LChoice` to a left-associative pattern. The result should be a sequential composition with the recursion variable as second argument and a strategy as first argument, where the recursion variable does not appear in the strategy. The second definition takes care of the presence of `CountRule` strategies, which are added by the frontend for profiling.

```
strategies

  Repeat(s1, s2) =
        {x: Rec(?x, LChoice(repeat(LAss);
                            Seq(not(oncetd(SVar(?x))); s1,
                                Call(SVar(?x), []))
                           ,not(oncetd(SVar(?x))); s2))}

  Repeat(s1, s2) =
        {x, s1', str:
         Rec(?x, LChoice(repeat(LAss);
                         ?Seq(s1', Seq(Call(SVar(x), []), Countrule(str)));
                         !Seq(Seq(s1', CountRule(str)), Call(SVar(x), []));
                         Seq(not(oncetd(SVar(?x))); s1, id)
                        ,not(oncetd(SVar(?x))); s2))}
```

Matching Only Traversals If a traversal is matching only it does not need to rebuild the subject term after inspection; the original subject term is not

transformed. This should be done for all traversals in general, but we try it out first for the oncetd traversal from the library.

```
strategies

  IsMatch = not(oncetd(Build(id)))

  Oncetd(s) = {x: Rec(?x, LChoice(s, One(Call(SVar(?x), [])))))}

  Cs = CsOncetd

rules

  CsOncetd ::
    Instr(Oncetd(?s; IsMatch), ?env, ?rcs) -->
    ! Block([
        Rpush(endloop),
        Label(startloop),
        Cpush(else),
        Instr(s, env, rcs),
        Cpop,
        Goto(repeat),
        Label(else),
        IsAppl,
        MatchTravInit,
        Label(nextson),
        OneMatchNextSon,
        Cpush(nextson),
        Rpush(doneit),
        Goto(startloop),
        Label(doneit),
        Cpop,
        MatchTravEnd,
        Label(repeat),
        Return,
        Label(endloop)
     ])
     where new => startloop;
           new => else;
           new => nextson;
           new => doneit;
           new => repeat;
           new => endloop
```

Fetch

```
strategies

  Fetch(s) = {x:
    Rec(?x, LChoice(Cong("Cons", [s, Id]),
```

```
                    Cong("Cons", [Id, Call(SVar(?x),[])]))))}

  Fetch(s) = {x:
    Rec(?x, LChoice(Seq(Cong("Cons",[s,Id]), CountRule("Cons")),
                    Seq(Cong("Cons",[Id,Call(SVar(?x),[])]), CountRule("Cons"))))}

  Cs = CsFetch

rules

  CsFetch ::
    Instr(Fetch(?s; IsMatch), ?env, ?rcs) -->
    ! Block([
        Tdupl,
        Label(a),
        MatchFun("Cons"),
        Cpush(b),
        Arg(0),
        Instr(s, env, rcs),
        Tpop,
        Cpop,
        Goto(end),
        Label(b),
        Arg(1),
        Tdrop,
        Goto(a),
        Label(end),
        Tpop,
      ])
   where new => a; new => b; new => end
```

This can be generalized to other traversals that have congruences with recursive calls, if the recursive call is only to one of the children and the others are identities.

5.3 Postproccesing

The postprocessing phase of the compiler takes an abstract machine program and derives initialization information from it. Currently it only deals with the initialization of rule counters by looking at the use of `CountRule` instructions.

5.3.1 postprocess

Postprocessing of generated code to extract global variable information.

```
module postprocess
imports instructions lib list-misc list-sort
```

```
rules

  CR1 : ICountRule(f) -> RuleCounter(f)

  CR2 : RuleCounter(f) -> RuleCounter(f, <new> f)

  CR3(b_rcs) : ICountRule(f) -> ICountRule(f')
        where b_rcs; fetch(match(RuleCounter(f, f')))

  main : Block(is) -> Program(RuleCounters(rcs), Block(is'))
         where
            <filter(CR1); uniq; map(CR2)> is => rcs;
            <map(try(CR3(build(rcs))))> is => is'
```

Chapter 6

Other Operations

6.1 Experiments

Consider as an example of optimization:

```
conc . map(s)
zip(id) . unzip -> id
```

Bibliography

[1] Peter Borovanský, Claude Kirchner, and Hélène Kirchner. Controlling rewriting by rewriting. In José Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, September 1996. Elsevier.

[2] Mark van den Brand and Eelco Visser. From Box to TEX: An algebraic approach to the generation of documentation tools. Technical Report P9420, Programming Research Group, University of Amsterdam, July 1994.

[3] 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.

[4] T.B. Dinesh, Magne Haveraaen, and Jan Heering. An algebraic programming style for numerical software and its optimization. CWI Report SEN-R9844, CWI, Amsterdam, The Netherlands, December 1998.

[5] Patricia Johann and Eelco Visser. Warm fusion in Stratego: A case study in the generation of program transformation systems. Technical report, Department of Computer Science, Universiteit Utrecht, 1999.

[6] Bas Luttik and Eelco Visser. Specification of rewriting strategies. In M. P. A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing, Berlin, November 1997. Springer-Verlag.

[7] P. A. Olivier and H. A. de Jong. Efficient annotated terms. Technical report, Programming Research Group, University of Amsterdam, August 1998.

[8] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[9] Eelco Visser. Strategic pattern matching. In *Rewriting Techniques and Applications (RTA'99)*, Lecture Notes in Computer Science, Trento, Italy, July 1999. Springer-Verlag. (To appear).

[10] Eelco Visser and Zine-el-Abidine Benaissa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, Electronic Notes in

Theoretical Computer Science, Pont-à-Mousson, France, September 1–4 1998. Elsevier.

[11] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, pages 13–26, Baltimore, Maryland, September 1998. ACM.

Appendix A

The Library

For the specification of transformation rules one usually needs all kinds of auxiliary functions to achieve local transformations. Examples are all kinds of operations on lists, renaming of bound variables and substitution. The strategy language comes with a library of predefined strategies for this purpose.

A.1 The Traversal Guide

A.1.1 SIMPLE-TRAVERSAL

```
module simple-traversal
imports basic
strategies
```

One Pass Traversal Apply s everywhere. All s applications have to succeed .

```
topdown(s)       = rec x(s; all(x))
bottomup(s)      = rec x(all(x); s)
downup(s)        = rec x(s; all(x); s)
downup2(s1, s2)  = rec x(s1; all(x); s2)
downup2'(s1, s2) = rec x((s1 <+ all(x)); s2)
```

Apply s everywhere along a spine.

```
spinetd(s) = rec x(s; try(one(x)))
spinebu(s) = rec x(try(one(x)); s)

spinetd'(s) = rec x(s; (one(x) + all(fail)))
spinebu'(s) = rec x((one(x) + all(fail)); s)
```

Apply s at one position. One s application has to succeed.

```
  oncetd(s) = rec x(s <+ one(x))
  oncebu(s) = rec x(one(x) <+ s)
```

Apply s at some positions, but at least one. As soon as one is found, searching is stopped, i.e., in the top-down case searching in subtrees is stopped, in bottom-up case, searching in upper spine is stopped.

```
  sometd(s) = rec x(s <+ some(x))
  somebu(s) = rec x(some(x) <+ s)
```

Find all topmost applications of `s`;

```
  alltd(s) = rec x(s <+ all(x))
```

Find as many applications as possible, but at least one.

```
  manybu(s) = rec x(some(x); try(s) <+ s)
  manytd(s) = rec x(s; all(try(x)) <+ some(x))

  somedownup(s) = rec x(s ; all(x) ; try(s) <+ some(x) ; try(s))
```

A.1.2 FIXPOINT-TRAVERSAL

Normalization Strategies: Fixed-point Traversal

```
module fixpoint-traversal
strategies

  reduce(s)     = repeat(rec x(some(x) + s))
  outermost(s)  = repeat(oncetd(s))
  innermost(s)  = repeat(oncebu(s))
  innermost'(s) = rec x(all(x) ; (s; x <+ id))

  innermost''(s) = {mark: where(new => mark);
                          rec x(@?(NF, mark) <+
                                all(x); (s; x <+ @!(NF, mark)))}

  pseudo-innermost3(s) =
        rec x(all(x); rec y(try(s; all(all(all(y); y); y); y)))

signature
  sorts Mark
  operations
    NF : Mark
```

A.2 Options and Tuples

A.2.1 OPTION

```
module option
```

```
signature
  sorts Option(a)
  operations
    None : Option(a)
    Some : a -> Option(a)
```

A.2.2 TUPLE

tindex: get the nth element of a tuple

tmap: apply a strategy to each element of a tuple

Tuple Concat: concatenate the lists in a tuple of lists, where the concatenation strategy s is a parameter.

```
module tuple
signature
  sorts Prod(ListType)
  operations
    TNil  : Prod([])
    TCons : a * Prod(lt) -> Prod(Cons(a, lt))
rules
  Fst   : TCons(x, tp) -> x
  Snd   : TCons(x, TCons(y, tp)) -> y
  Third : TCons(x, TCons(y, TCons(z, tp))) -> z
  TInd1 : (1, TCons(x, tp)) -> x
  TInd2 : (n, TCons(x, tp)) -> (<minus> (n, 1), tp)

  Dupl  : x -> (x, x)
strategies

  tindex = rec x(TInd1 <+ TInd2 ; x)

  tmap(s) = rec x(TNil + TCons(s, x))

  tconcat(s) = rec y(<<TNil -> Nil>>
                   + {x, xs: <<TCons(x, xs) -> (x, <y> xs)>>} ; s)

  at_tsuffix(s) = rec x(s <+ TCons(id, x))

  tcata(s1, s2) = rec y(TNil; s1 <+
                        {x, xs: <<TCons(x, xs) -> (x, <y> xs)>>}; s2)
```

A.3 Lists

A.3.1 LIST

```
module list
imports simple-traversal
```

```
        tuple
        list-cons
        list-basic
        list-index
        list-zip
        list-sort
        list-set
        list-lookup
        list-misc
```

A.3.2 LIST-CONS

```
module list-cons
signature
  sorts List(Type)
  operations
    Nil  : List(a)
    Cons : a * List(a) -> List(a)
```

A.3.3 LIST-BASIC

Basic functionality on lists. Constructors for lists are defined in list-cons.

Map: Apply strategy to each element of a list

Length of a list

Fetch: Find first list element for which s succeeds

At tail: apply a strategy to the tail of a list

At suffix: apply a strategy to some suffix of a list

```
module list-basic
imports list-cons
rules
  Hd     : Cons(x, l) -> x
  Tl     : Cons(x, l) -> l
  Last   : Cons(x, Nil) -> x
strategies

  map(s) = rec x(Nil + Cons(s, x))

  list(s) = rec x(Nil + Cons(s, x))

  length = rec x(Nil; build(0) + Tl; x; {n:<<n -> <plus> (n, 1)>>})

  fetch(s) = rec x(Cons(s, id) <+ Cons(id, x))

  at_tail(s) = rec x(Cons(id, x) + Nil ; s)
```

```
  at_end(s) = rec x(Cons(id, x) + Nil ; s)

  at_suffix(s) = rec x(s <+ Cons(id, x))

  at_last(s) = rec x(Cons(id, Nil); s <+ Cons(id, x))

  listbu(s)       = rec x((Nil + Cons(id, x)); s)
  listtd(s)       = rec x(s; (Nil + Cons(id, x)))
  listdu(s)       = rec x(s; (Nil + Cons(id, x)); s)
  listdu2(s1, s2) = rec x(s1; (Nil + Cons(id, x)); s2)

rules

  RevInit : xs -> (xs, [])
  Rev     : (Cons(x, xs), ys) -> (xs, Cons(x, ys))
  RevExit : ([], ys) -> ys

strategies

  reverse = RevInit; repeat(Rev); RevExit

rules
  UptoInit : i -> (i, [])
  UptoExit : (0, xs) -> xs
  UptoStep : (i, xs) -> (<minus> (i, 1), Cons(i, xs))
strategies
  upto = UptoInit; rec x(UptoExit <+ UptoStep; x)

rules
  conc : (l1, l2) -> <at_tail(build(l2))> l1
  Concat(x) : Cons(l, ls) -> <at_end(build(<x> ls))> l
strategies
  concat = rec y(Nil + Concat(y))
```

A.3.4 LIST-INDEX

```
module list-index
imports list-cons simple-traversal
rules

  Ind1   : (1, Cons(x, xs)) -> x
  Ind2   : (n, Cons(x, xs)) -> (<min> (n, 1), xs) where <geq> (n, 2)

  Gind0  : (x, ys) -> (1, x, ys)
  Gind1  : (n, x, Cons(x, xs)) -> n
  Gind2  : (n, y, Cons(x, xs)) -> (<plus> (n, 1), y, xs)

strategies
```

```
(* Index: Get the n-th element of a list *)

  index = repeat(Ind2) ; Ind1

(* Get-index: get index of element in list *)

   get_index = Gind0 ; rec x(Gind1 <+ Gind2 ; x)
```

A.3.5 LIST-LOOKUP

Lookup: find value associated to key

Find first element of a list to which a strategy s applies

An alternative formulation of lookup is the following. The advantage over lookup is that it does not construct intermediate pairs.

```
module list-lookup
imports list-basic
rules

  Look1  : (x, Cons((x, y), _)) -> y
  Look2  : (x, Cons((_, _), xs)) -> (x, xs)

strategies

   lookup = rec x(Look1 <+ Look2 ; x)

   getfirst(s) = rec x(Hd ; s <+ Tl ; x)

   lookup' = {x, xs: match((x, xs)) ; <getfirst({y:<<(x, y) -> y>>})> xs}
```

A.3.6 LIST-MISC

```
module list-misc
imports list-cons
imports list-basic
strategies

(* Member *)

  member = {x : (match(x), fetch(match(x)))}

(* Filter *)

  filter(s) = rec x(Nil + (Cons(s, x) <+ Tl ; x))

(* Fold *)

rules
```

```
  FoldR    : Cons(x, xs) -> (x, xs)
  TFoldR   : TCons(x, xs) -> (x, xs)

  FoldL(s) : (Cons(x, xs), y) -> (xs, <s> (x, y))
  FoldL(s) : (Nil, y) -> y

strategies

  foldr(s1, s2)  = rec x(Nil; s1 + FoldR; (id, x); s2)
  tfoldr(s1, s2) = rec x(TNil; s1 + TFoldR; (id, x); s2)

  foldl(s) = repeat(FoldL(s))

(* MapFold *)

  mapfoldr(s1, s2, s3) = rec x(Nil ; s1 <+ kids ; (s2, x) ; s3)

(* The last element in a list *)

  last = rec x(Last <+ Tl ; x)
```

A.3.7 LIST-SET

Union: Concatenation of two lists, only those elements in the first list are added that are not already in the second list.

Diff: Difference of two lists

```
module list-set
imports list-basic

rules

  HdMember(mklst) : Cons(x, xs) -> xs where mklst; fetch(match(x))

  HdMember'(eq, mklst) :
     Cons(x, xs) -> xs
     where mklst; fetch({y: ?y; <eq> (x, y)})

strategies

  union =
  {l1, l2:
   <<(l1, l2) -> l1>>;
   rec x(  Nil; !l2
        <+ HdMember(!l2); x
        <+ Cons(id, x)
        )
  }
```

```
  diff =
  {l1, l2:
   <<(l1, l2) -> l1>>;
   rec x(  Nil
        <+ HdMember(!l2); x
        <+ Cons(id, x)
        )
  }

  diff'(eq) =
  {l1, l2:
   <<(l1, l2) -> l1>>;
   rec x(  Nil
        <+ HdMember'(eq, !l2); x
        <+ Cons(id, x)
        )
  }

  isect = {l1, l2: ?(l1, l2); <diff> (l1, <diff> (l1, l2))}


imports tuple
strategies
  collect(s) = rec x(s; {y: ?y; ![y]} <+ kids; tmap(x); tconcat(union))

  nrofoccs(s) = rec count(s; build(1) <+
                          kids; tmap(count); tcata(build(0), plus))

  twicetd(s) = oncetd(kids; at_tsuffix(TCons(oncetd(s), oncetd(s))))
```

problem for transformation: does not put back constructor that is deleted by kids. There should be a way to do something to the kids of a constructor consecutively.

```
  atmostonce(s) = not(twicetd(s))

  atmostonce'(s) = {n : nrofoccs(s) => n; <leq> (n, 1)}
```

A.3.8 LIST-SORT

(* Sorting *)

```
module list-sort
imports list-basic
rules

  Sort(s) : Cons(x, Cons(y, l)) -> Cons(y, Cons(x, l))
            where <s> (x, y)
```

```
  LSort(s) : Cons(x, l) -> Cons(y, Cons(x, l'))
             where <at_suffix(<<Cons(y, ys) -> ys where <s> (x, y)>>)> l => l'

  Uniq : Cons(x, l) -> l
         where <fetch(match(x))> l

strategies

  sort(s) = try(rec x((s <+ Cons(id, x)) ; try(x)))

  isort(s) = try(rec x((Cons(id, x) <+ s) ; try(x)))

  uniq = listbu(repeat(Uniq))
```

A.3.9 LIST-ZIP

Zip and Unzip

```
module list-zip
imports list-cons
imports tuple
rules

  Zip1   : (Nil, Nil) -> Nil
  Zip1a' : (Nil, _) -> Nil
  Zip1b' : (_, Nil) -> Nil
  Zip2   : (Cons(x, xs), Cons(y, ys)) -> ((x, y), (xs, ys))
  Zip3   : (x, xs) -> Cons(x, xs)

  UnZip1 : Nil -> (Nil, Nil)
  UnZip2 : ((x, y), (xs, ys)) -> (Cons(x, xs), Cons(y, ys))
  UnZip3 : Cons(x, xs) -> (x, xs)

  NZip0 : xs -> (1, xs)
  NZip1 : (n, Nil) -> Nil
  NZip2 : (n, Cons(y, ys)) -> ((n, y), (<plus> (n, 1), ys))
  NZip3 : (x, xs) -> Cons(x, xs)

  TZip1   : (TNil, TNil) -> TNil
  TZip2   : (TCons(x, xs), TCons(y, ys)) -> ((x, y), (xs, ys))
  TZip3   : (x, xs) -> TCons(x, xs)

  cart(s) : (xs, ys) ->
            <map({x: ?x; <map({y: ?y; <s>(x, y)})> ys}); foldr(![], union)> xs

strategies

  genzip(a, b, c, s) = rec x(a + b ; (s, x) ; c)
```

```
  zip(s)  = genzip(Zip1,   Zip2,   Zip3,   s)
  zip'(s) = genzip(Zip1a' <+ Zip1b',   Zip2,   Zip3,   s)
  unzip   = genzip(UnZip1, UnZip3, UnZip2, id)
  nzip(s) = NZip0 ; genzip(NZip1,  NZip2,  NZip3,  s)
  tzip(s) = genzip(TZip1,   TZip2,   TZip3,   s)

  zipFetch(s) = rec x(Zip2; ((s, id) <+ (id, x)))
```

A.4 Strings

A.4.1 STRING

```
module string
imports list-basic
strategies

  conc-strings = (explode-string, explode-string); conc; implode-string

  concat-strings = map(explode-string); concat; implode-string
```

A.5 Text

A.5.1 TEXT

In this module we define a simple algebra of texts. Texts are composed of a vertical composition of lines, which are lists of integers representing the (ASCII) code of characters.

```
module text
imports lib
signature
  sorts Text
  operations
    Cs : List(Int)   -> Line
    L  : Line        -> Text
    V  : Text * Text -> Text
    E  : Text
rules

  PP-V  : V(V(x, y), z) -> V(x, V(y, z))

  PP-V  : V(E, x) -> x
  PP-V  : V(x, E) -> x
```

Simple operations on texts

`Wd(i)` creates blank text with width `i`

```
signature
  operations
    Wd  : Int         -> Line
    Lwd : Line        -> Line
    LC  : Line * Line -> Line
    LP  : Line * Text -> Text
    LI  : Line * Text -> Text

rules

  PP-Wd : Wd(i) -> Cs(xs)
          where <upto; map(!32)> i => xs

  PP-LP : LP(x, E) -> E

  PP-LP : LP(x, L(y)) -> L(LC(x, y))

  PP-LP : LP(x, V(y, z)) -> V(LP(x, y), LP(x, z))

  PP-LI : LI(x, E) -> L(x)

  PP-LI : LI(x, L(y)) -> L(LC(x, y))

  PP-LI : LI(x, V(L(y), z)) ->
          V(L(LC(x, y)), LP(Lwd(x), z))

  PP-Lw : Lwd(Cs(xs)) -> Cs(<map(!32)> xs)

  PP-LC : LC(Cs(xs), Cs(ys)) -> Cs(<conc> (xs, ys))

  PP-LC : LC(L(x), y) -> LC(x, y)

        (* NOTE: this should not be necessary *)
```

Rules for operators: strings, horizontal composition and indentation

```
signature
  operations
    S  : String      -> Text
    H  : Text * Text -> Text
    I  : Text        -> Text
rules

  PP-S : S(s) -> L(Cs(<explode-string> s))

  PP-S' : L(Cs(cs)) -> S(<implode-string> cs)

  PP-I : I(x) -> LP(Wd(2), x)

  PP-H : H(L(x), y) -> LI(x, y)
```

```
  PP-H : H(V(x, y), z) -> V(x, H(y, z))

signature
  operations
    Hs : List(Text) -> Text
    Vs : List(Text) -> Text
rules

  PP-Hs : Hs([]) -> E

  PP-Hs : Hs(Cons(x, y)) -> H(x, Hs(y))

  PP-Vs : Vs([]) -> E

  PP-Vs : Vs(Cons(x, y)) -> V(x, Vs(y))

strategies

  normalize-text =
        pseudo-innermost3(
                PP-V + PP-LP + PP-LI + PP-S + PP-I + PP-H +
                PP-Lw + PP-LC + PP-Wd + PP-Hs + PP-Vs)

  text-nf = rec txt(V(txt, txt) +
                    rec ln(H(ln, ln) + L(Cs(id)) + Wd(id)))

  text-nf = rec txt(V(L(Cs(id)), txt) + L(Cs(id)))
```

A.6 The Entire Library

All modules in the library are imported in `lib`.

A.6.1 LIB

```
module lib
imports basic
        simple-traversal
        fixpoint-traversal
        list
        tuple
        option
```