The Stratego Compiler

Eelco Visser

STRATEGO

For Stratego version 0.4.17

www.stratego-language.org

Department of Information and Computing Sciences
Universiteit Utrecht
August 27, 2000

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

The Stratego Compiler

Eelco Visser

August 27, 2000

Summary

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

CONTENTS

Chapter: 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.

Chapter: A Bootstrapped Compiler for Strategies
Section: 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 [4] to define language processors, in particular program transformations. For example: normalization of box expressions for pretty-printing [3]; normalization of syntax definitions [8]; transformation of C++ programs [5].

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 [2], include operators for sequential non-deterministic 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 definining 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 approach 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 [**?**] 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.

Chapter: A Bootstrapped Compiler for Strategies
Section: 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].

```
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: Specification of several generic term traversal strategies.

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.

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.

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 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 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.

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 the modules in Figure 2. 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 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.

```
module terms
imports list-cons
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) *)

module strategy
imports terms
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 (* ?t *)
    Build    : Term                         -> Strat (* !t *)

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

    Prim    : String -> Strat
```

Figure 2: Abstract syntax of terms and System S expressions.

LIBRARY

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

- Simple traversals (such as in Figure 1)
- Fixed-point traversals
- 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. Note that this concerns *object* variables in the language being manipulated, which are different from the meta-variable used in rules.

Chapter: A Bootstrapped Compiler for Strategies
Section: 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.

Chapter: A Bootstrapped Compiler for Strategies
Section: The Compiler

ARCHITECTURE

The overall architecture of the compiler is shown in Figure 3. The compiler consists of four main components: front-end, optimizer, matching-tree, and back-end.

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.

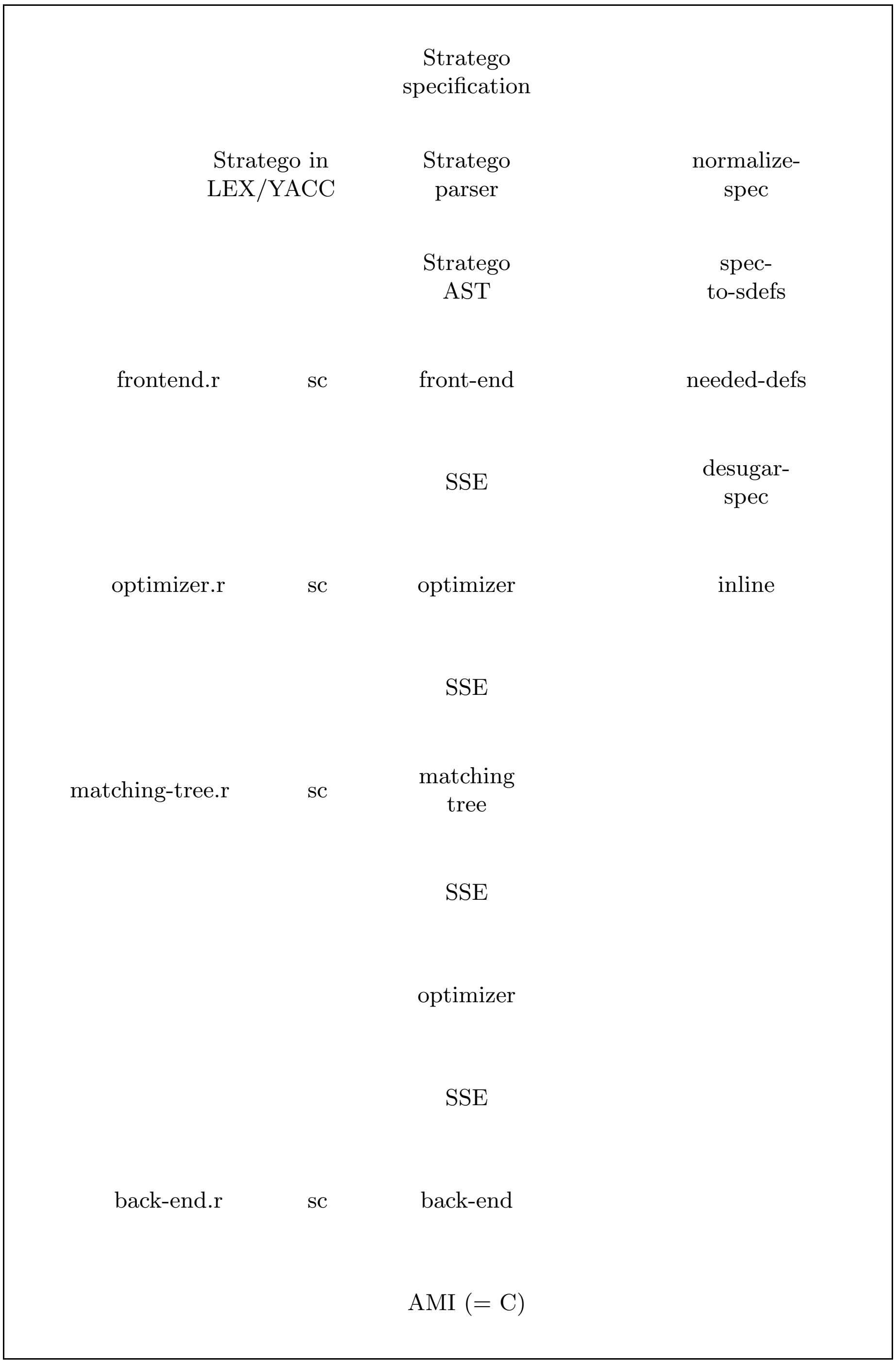

Figure 3: Architecture of the Stratego compiler (sc)

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 representation of terms the ATerms package [7] is used. This package provides an implementation of terms based on hash-consing and supports garbage collection.

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.

Chapter: A Bootstrapped Compiler for Strategies
Section: Examples

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

Chapter: A Bootstrapped Compiler for Strategies
Section: Examples

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.

Note that in general such a pipeline may fail to apply to a term. In this case, if an error is detected either in the use-def analysis (due to undeclared or unitialized variables) or in the needed-defs transformation (missing definition), the pipeline fail. In these cases an error message is derived from the failure.

```
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
```

Chapter: A Bootstrapped Compiler for Strategies
Section: Examples

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 right-hand side replaces the build by a scope construct that declares a new local variable x. Inside the scope first s is applied to t’ and the result matched against the new variable x. This variable is then used inside the term t in the build.

The 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.

```
module desugar
 ...
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))
```

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`.

```
module compiler
 ...
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. After translation the nested code is flattened to a single list. Some simple optimizations are performed by `peephole` and finally, the code is wrapped in some support code by `Assemble`.

```
module backend
  ...
strategies

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

Part I

Basics

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
imports list-cons
signature
  sorts Term
  constructors
    Wld  :                          Term   (* _ *)
    Var  : String                -> Term   (* x *)
    Int  : Int                   -> Term   (* 0, 1, 2 ... *)
    Real : Real                  -> Term   (* 0.0, 0.1, 2.3 ... *)
    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))

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
  constructors
    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 *)
    Thread  : Strat                        -> Strat    (* thread(s) *)
```

MATCHING AND BUILDING

```
signature
  constructors
    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
  constructors
    Prim    : String -> Strat

    CountRule : String -> Strat
```

SIGNATURES

Types and Kinds

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

Signatures

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

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 .

```
module sugar
imports strategy signatures
```

VARIADIC VERSIONS OF SOME COMBINATORS

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

MATCH, BUILD, APPLY COMBINATIONS

```
signature
  constructors
    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
  constructors
    Con          : Var * Term * SVar -> Term    (* x[t](f) *)
    App          : Strat * Term      -> Term    (* <s> t *)
    Explode      : Term * Term       -> Term    (* t1 # (t2) *)
    ExplodeCong  : Strat * Strat     -> Strat   (* s1 # (s2) *)
    As           : Var * Term        -> Term    (* x@t *)
    BuildDefault : Term              -> Term    (* _ t *)
```

RULES & RULE DEFINITIONS

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

OVERLAYS

```
signature
  sorts Overlay
  constructors
    Overlay : String * List(String) * Term -> Overlay
```

SPECIFICATIONS

```
signature
  sorts BSpec Spec
  constructors
    Signature     : List(BSig)     -> BSpec
    Overlays      : List(Overlay) -> BSpec
    Rules         : List(RDef)     -> BSpec
    Strategies    : List(SDef)     -> BSpec
    Imports       : List(String)   -> BSpec
    Specification : List(BSpec)    -> Spec
```

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))
```

```
%{

#include <aterm2.h>
#include "stratego.h"
#include "stratego.grm.tab.h"
#include "options.h"

static void eat_comment();
static void eat_c_comment();
static ATerm mkstring();
void yyerror(char *msg);

FILE *outfile;

#define lexerror(x, y) fprintf(stderr, "%s %s\n", x, y)

#define YY_DECL  int yylex(YYSTYPE *lvalp)

extern char file_name[256];

/*
lcid {lcstart}{idchars}*
ucid {ucstart}{idchars}*
lcid {idchars}+
ucid {idchars}+
*/

%}

lcstart [a-z‘]
ucstart [A-Z_]
idchars {lcstart}|{ucstart}|[0-9’.\-]
lcid {lcstart}{idchars}*
ucid {ucstart}{idchars}*
primeid [’]{lcid}|{ucid}
ID {lcid}|{ucid}
ws [\t\ ]*
num [0-9]+
real [\-]?[0-9]+[.][0-9]+
shortcom "//"[^\n]*[\n]
backslash \x5C
literate {backslash}"literate"
endcode {backslash}"end\{code\}"
begincode {backslash}"begin\{code\}"
endliterate {backslash}"endliterate"

%x COMM MODNAME IMP LIT STARTLIT

%option yylineno
%option noyywrap
```

```
%%

/* Literate programs */

<INITIAL>{literate} BEGIN LIT;
<INITIAL>{endcode} BEGIN LIT;
<LIT>{begincode} BEGIN INITIAL;
<LIT>{endliterate} BEGIN INITIAL;
<LIT>\n ; /* ignore newlines */
<LIT>. ; /* ignore text */

/* Layout */

<INITIAL>{ws} ; /* ignore whitespace */
<INITIAL>{shortcom}  ; /* ignore comments */
<INITIAL>\n ; /* ignore newlines */

/* Comments (SML style) */

<INITIAL>"(*"   eat_comment();
<INITIAL>"/*"   eat_c_comment();

/* Module header */

<INITIAL>"module" BEGIN MODNAME; return MODULE;
<INITIAL>"*" return ASTERISK;
<INITIAL>"=" return EQ;

/* Imports */

<INITIAL,MODNAME>"imports"  BEGIN MODNAME; return IMPORTS;

<INITIAL,MODNAME>"strategies"  BEGIN INITIAL; return STRATEGIES;
<INITIAL,MODNAME>"rules"       BEGIN INITIAL; return RULES;
<INITIAL,MODNAME>"signature"  BEGIN INITIAL; return SIGNATURE;
<INITIAL,MODNAME>"overlays"  BEGIN INITIAL; return OVERLAYS;

<MODNAME>"(*" eat_comment();
<MODNAME>"/*" eat_c_comment();

<MODNAME>{endcode} BEGIN LIT;

<MODNAME>{lcid}  (*lvalp).term = ATmakeString(yytext); return LCID;
<MODNAME>{ucid}  (*lvalp).term = ATmakeString(yytext); return UCID;

<MODNAME>[ \t\n] ; /* ignore whitespace */

  /* keywords */
```

```
<INITIAL>"rec" return MU;
<INITIAL>"fail" return FAIL;
<INITIAL>"id" return SUCC;

<INITIAL>"some" return SOMETOK;
<INITIAL>"one" return ONE;
<INITIAL>"thread" return THREAD;
<INITIAL>"all" return ALL;
<INITIAL>"not" return NOT;

<INITIAL>"?"      return MATCHe;
<INITIAL>"!"  return BUILDe;

<INITIAL>"@"  return AS;

<INITIAL>"@?"      {/*yymessage("@? not implemented"); */ return ANNMATCH;}
<INITIAL>"@!"  {/*yymessage("@! not implemented"); */ return ANNBUILD;}
<INITIAL>"@/"  {yymessage("@/ not implemented"); return ANNRM;}

<INITIAL>"where"  return WHERE;
<INITIAL>"test"  return TEST;
<INITIAL>"sorts"  return SORTS;

<INITIAL>"constructors" return OPERATIONS;
<INITIAL>"prim"  return PRIM;


<INITIAL>"match"  {yymessage("match obsolete; use ?"); return MATCH;}
<INITIAL>"build"  {yymessage("build obsolete; use !"); return BUILD;}
<INITIAL>"operations"  {yymessage("'operations' obsolete; use 'constructors'");
 return OPERATIONS;}


/* Operators */

<INITIAL>"\\"     return BACKSLASH;
<INITIAL>"."     return DOT;
<INITIAL>":-" return BACKARROW;
<INITIAL>":=" return ASSIGN;
<INITIAL>"+" return PLUS;
<INITIAL>"<+" return LTPLUS;
<INITIAL>"_" return UNDERSCORE;
<INITIAL>"**" return STARSTAR;
<INITIAL>"*" return ASTERISK;
<INITIAL>"=" return EQ;
<INITIAL>"," return COMMA;
<INITIAL>";" return SEMICOLON;
<INITIAL>":" return COLON;
<INITIAL>"::" return DOUBLECOLON;
```

```
<INITIAL>"(" return LPAREN;
<INITIAL>")" return RPAREN;
<INITIAL>"{" return LCURLY;
<INITIAL>"}" return RCURLY;
<INITIAL>"[" return LBRACK;
<INITIAL>"]" return RBRACK;
<INITIAL>"<" return LT;
<INITIAL>">" return GT;
<INITIAL>"<<" return LL;
<INITIAL>">>" return GG;
<INITIAL>"-->" return LONGARROW;
<INITIAL>"->" return ARROW;
<INITIAL>"=>" return DOUBLEARROW;
<INITIAL>"<=" return LEFTDOUBLEARROW;
<INITIAL>"|" return BAR;
<INITIAL>"#"      return EXPLODE;
<INITIAL>"^"      return MODIFIER;

/* Identifiers, strings and numbers */

<INITIAL>{lcid}  (*lvalp).term = ATmakeString(yytext); return LCID;
<INITIAL>{ucid}  (*lvalp).term = ATmakeString(yytext); return UCID;
<INITIAL>{primeid}  (*lvalp).term = ATmakeString(yytext+1); return UCID;
<INITIAL>"\""           (*lvalp).term = mkstring(); return STRINGTOK;
<INITIAL>{num} (*lvalp).num  = atoi(yytext); return INT;
<INITIAL>{real} (*lvalp).real = atof(yytext); return REAL;

/* All other characters are wrong */

<INITIAL>. lexerror ("ignoring illegal character", yytext);

<<EOF>>{BEGIN(INITIAL); yyterminate();}

%%

/*
 * Function which skips comment chars. The function allows '*' to
 * occure within comments.
 */

static void eat_comment()
{
   char c;

   while( 1 )
   {
      /* get a character */
      c = input();

      /* if it equals '*" and the next character equals ')' then
```

```
       * the end of the comment is reached.
       */
      if( c == ’*’ )
      {
         c = input();
         if( c == ’)’ )
            return;
      }
   }
}


static void eat_c_comment()
{
   char c;

   while( 1 )
   {
      /* get a character */
      c = input();

      /* if it equals ’*" and the next character equals ’/’ then
       * the end of the comment is reached.
       */
      if( c == ’*’ )
      {
         c = input();
         if( c == ’/’ )
            return;
      }
   }
}

static char* resize_buf( char* buf, int size)
{
   buf = (char*)realloc( buf, size );
   assert( buf != NULL );
   return buf;
}

#define CHECK(buf, size, ind ) \
if( ind == size ){ size *= 2; buf = resize_buf( buf, size );}

/*
 * This functions parses a quoted string and accepts
 * escaped quotes (e.g., "hello \"world\"") within strings.
 */
static ATerm mkstring()
{
   ATerm theString;
```

```
char* buf = NULL;
int c;
int size = 512;
int ind  = 0;

/* use dynamically resizable buffer to store string */
buf = resize_buf( buf, size );

/* add opening quote */
CHECK( buf, size, ind );
buf[ind++] = ’"’;

while( 1 )
{
   c = input();

   if( c == ’\"’ )
      break;

   if( c <= 0 )
   {
      yyerror( "String not terminated at eof" );
      return NULL;
   }

   /* Handle escaped character */
   if( c == ’\\’ )
   {
      CHECK( buf, size, ind );
      buf[ind++] = c;
      c = input();
   }

   CHECK( buf, size, ind );
   buf[ind++] = c;
}
/* Add closing quote */
CHECK( buf, size, ind );
buf[ind++] = ’"’;

/* terminate string */
buf[ind] = ’\0’;

/* convert to aterm */
theString = ATparse( buf );

/* free buffer */
free( buf );

/* return quotes string */
```

```
   return theString;
}
/* flex user code */
```

```
%{
#include <aterm1.h>
#include "stratego.h"

extern int yylineno;
extern char file_name[256];
extern FILE *yyin;

ATerm parse_tree;

void yymessage(char *msg)
{
  fprintf(stderr, "%s: line %d - %s\n", file_name, yylineno + 1, msg);
}

void yyerror(char *msg)
{
  yymessage(msg);
  exit(1);
}

%}

%union{
   int    num;
   double real;
   char   *string;
   ATerm  term;
   ATermList list;
}

%{

/* int yylex(void); */

int yylex(YYSTYPE *lvalp);
int yyparse(void);

int parse()
{
   int result;
   result = yyparse();
   return result;
}
%}

%token <term> LCID
%token <term> STRINGTOK
%token <term> UCID
%token <term> ID
```

Chapter: Parser
Section: Parser

```
%token ARROW
%token LONGARROW
%token ASSIGN
%token ASTERISK
%token BACKARROW
%token BAR
%token BUILD
%token AS
%token ANNBUILD
%token ANNRM
%token BUILDe
%token COMMA
%token EQ
%token FAIL
%token GG
%token IMPORTS
%token KIDS
%token LBRACK
%token LCURLY
%token LL
%token LPAREN
%token LT
%token GT
%token ANNMATCH
%token MATCH
%token MATCHe
%token MODULE
%token NEW
%token OPERATIONS
%token OVERLAYS
%token PARSEPROG
%token PARSEQUERY
%token PRIM
%token RCURLY
%token RPAREN
%token RULES
%token SIGNATURE
%token SORTS
%token STARSTAR
%token STRATEGIES
%token STR_GT
%token SUCC
%token UNDERSCORE
%token DOT
%token BACKSLASH
%token EXPLODE
%token MODIFIER

%right COLON DOUBLECOLON
```

Chapter: Parser
Section: Parser

```
%right PLUS LTPLUS
%right SEMICOLON
%right DOUBLEARROW
%right LEFTDOUBLEARROW
%left  ASSIGN
%right RBRACK
%right NOT WHERE TEST ONE ALL THREAD SOMETOK MU
%right <real> REAL
%right <num> INT

%type <term> decl
%type <list> decls
%type <term> id
%type <list> idlist
%type <term> kind
%type <list> kinds
%type <list> mods
%type <term> opdecl
%type <list> opdecls
%type <term> optcond
%type <list> optcont
%type <term> optkind
%type <list> optstrategylist
%type <list> opttermlist
%type <list> optvarlist
%type <list> overlays
%type <term> overlay
%type <term> rule
%type <term> stratrule
%type <term> rule_def
%type <list> rules
%type <term> sdecl
%type <list> sdecls
%type <term> start
%type <list> strategies
%type <term> strategy
%type <term> optapplication
%type <term> strategy_def
%type <list> strategylist
%type <term> strategytail
%type <term> tail
%type <term> term
%type <list> termlist
%type <term> trav
%type <list> tvarlist
%type <term> type
%type <list> typelist
%type <list> varlist

%start start
```

```
%pure_parser

%%

start  : MODULE id decls          {parse_tree = ATmake("Specification([<list>])", $3);}
        | LBRACK strategy RBRACK term  {parse_tree = App2("Trm", $2, $4);}
| strategy {parse_tree = App1("Strategy",$1);}
;

decls : decl       {$$ = ATmakeList1($1);}
| decls decl  {$$ = ATappend($1, $2);}
  | {$$ = ATmakeList0();}
        ;

mods :  {$$ = ATmakeList0();}
| mods id {$$ = ATappend($1, $2);}
;

decl : IMPORTS mods {$$ = App1("Imports", (ATerm) $2);}
| RULES rules  {$$ = App1("Rules", (ATerm) $2);}
| STRATEGIES strategies  {$$ = App1("Strategies", (ATerm) $2);}
| SIGNATURE sdecls {$$ = App1("Signature", (ATerm) $2);}
| OVERLAYS overlays {$$ = App1("Overlays", (ATerm) $2);}
        ;

sdecls : sdecl {$$ = ATmakeList1($1);}
| sdecls sdecl  {$$ = ATappend($1, $2);}
|  {$$ = ATmakeList0();}
        ;

sdecl : SORTS idlist {$$ = App1("Sorts", (ATerm) $2);}
| OPERATIONS opdecls {$$ = App1("Operations", (ATerm) $2);}
        ;

idlist  :                    {$$ = ATmakeList0();}
        | idlist id optkind             {$$ = ATinsert($1, App2("Sort", $2, $3));}

optkind :  {$$ = ATmake("Nokind");}
| LPAREN termlist RPAREN  {$$ = App1("Kinds", (ATerm) $2);}
        ;

kinds   : kind  {$$ = ATmakeList1($1);}
| kinds kind  {$$ = ATappend($1, $2);}
        ;

kind : ASTERISK {$$ = ATmake("Type");}
| STARSTAR {$$ = ATmake("TypeList");}
        ;
```

```
opdecls :  {$$ = ATmakeList0();}
| opdecls opdecl  {$$ = ATappend($1, $2);}
        ;

opdecl : id COLON type {$$ = App2("OpDecl", $1, $3);}
        ;

type  : typelist ARROW term  {$$ = App2("FunType", (ATerm) $1, $3);}
| term {$$ = App1("ConstType", $1);}
        ;

typelist
: typelist ASTERISK term  {$$ = ATappend($1, $3);}
| term {$$ = ATmakeList1($1);}
        ;

id      : LCID  {$$ = $1;}
    | UCID  {$$ = $1;}
        ;

/* Terms */

term  : id optcont        {if(ATisEmpty($2))
   $$ = App1("Var", $1);
 else
   $$ = App3("Con",
      App1("Var", $1), ATgetFirst($2),
      ATgetFirst(ATgetNext($2)));}

| UNDERSCORE {$$ = ATmake("Wld");}

| UNDERSCORE term {$$ = App1("BuildDefault", $2);}

| INT        {$$ = App1("Int", (ATerm) ATmakeInt($1));}
| REAL        {$$ = App1("Real", (ATerm) ATmakeReal($1));}
| STRINGTOK {$$ = App1("Str", $1);}
      | id opttermlist  {$$ = App2("Op", $1, (ATerm) $2);}
        | id AS term                    {$$ = App2("As", App1("Var", $1), $3);}
| LT strategy GT term  {$$ = App2("App", $2, $4);}
| LBRACK termlist tail RBRACK   {$$ = list_to_consnil_op_tl($2, $3);}
| LPAREN termlist RPAREN {$$ = list_to_tconstnil_op($2);}
| term EXPLODE LPAREN term RPAREN
{$$ = App2("Explode", $1, $4)}
        ;

tail    :  {$$ = ATmake("Op(\"Nil\",[])");}
| BAR term  {$$ = $2;}
        ;

optcont : {$$ = ATmakeList0();}
```

```
| LBRACK term optcond RBRACK trav
{$$ = ATmakeList2($2, App2("Call", $5, (ATerm) ATmakeList0()));}
        ;

trav    :  {$$ = ATmake("SVar(\"oncetd\")"); }
| LPAREN id RPAREN  {$$ = App1("SVar", $2);}

opttermlist
:  {$$ = ATmakeList0();}
| LPAREN termlist RPAREN  {$$ = $2;}
        ;

termlist: term {$$ = ATmakeList1($1);}
| term COMMA termlist   {$$ = ATinsert($3, $1);}
        | {$$ = ATmakeList0();}
        ;

tvarlist: LCID {$$ = ATmakeList1($1);}
| LCID COMMA tvarlist   {$$ = ATinsert($3, $1);}
        | {$$ = ATmakeList0();}
        ;

/* Rewrite rules */

rules  :            {$$ = ATmakeList0();}
       | rules rule_def   {$$ = ATappend($1, $2);}
        ;

rule_def
: id optvarlist COLON rule {$$ = App3("RDef", $1, (ATerm) $2, (ATerm) $4);}
| id optvarlist
DOUBLECOLON stratrule {$$ = App3("RDef", $1, (ATerm) $2, (ATerm) $4);}
        ;

rule    : term ARROW term optcond  {$$ = App3("Rule", $1, $3, $4);
}
        ;

stratrule : strategy LONGARROW strategy optcond
{$$ = App3("StratRule", $1, $3, $4); }
        ;

optcond : {$$ = ATmake("Id");}
| WHERE strategy {$$ = App1("Where", $2);}
        ;

/* Strategies */

optapplication
:  {$$ = ATmake("Id");}
```

```
| LEFTDOUBLEARROW strategy {$$ = $2;}

strategy
: id optstrategylist  {$$ = App2("Call", App1("SVar", $1), (ATerm) $2);}

| id MODIFIER id optstrategylist
  {$$ = App2("Call", App1("SVar", App2("Mod", $1, $3)), (ATerm)$4);}

| strategy EXPLODE LPAREN strategy RPAREN
{$$ = App2("ExplodeCong", $1, $4)}

| MATCH LPAREN term RPAREN {$$ = App1("Match", $3);}

| MATCH LPAREN term RPAREN {$$ = App1("Match", $3);}

| MATCHe term optapplication {$$ = App2("AM", $3, $2);}

/* | MATCHe term  {$$ = App1("Match", $2);} */

| BUILD  LPAREN term RPAREN {$$ = App1("Build", $3);}
| BUILDe term {$$ = App1("Build", $2);}

| ANNMATCH LPAREN term COMMA term RPAREN
{$$ = App2("AnnMatch", $3, $5);}
| ANNBUILD LPAREN term COMMA term RPAREN
{$$ = App2("AnnBuild", $3, $5);}
| ANNRM term                    {$$ = App1("AnnRemove", $2);}

| NEW     {$$ = ATmake("Prim(\"new\")");}
| STR_GT     {$$ = ATmake("Prim(\"str_gt\")");}
| KIDS     {$$ = ATmake("Prim(\"kids\")");}
        | PRIM LPAREN STRINGTOK RPAREN  {$$ = App1("Prim", $3);}

        | LL rule GG {$$ = App1("SRule", $2);}
        | BACKSLASH rule BACKSLASH {$$ = App1("LRule", $2);}
| LT strategy GT term {$$ = App2("BA", $2, $4);}

| strategy DOUBLEARROW term {$$ = App2("AM", $1, $3);}

/* | term ASSIGN term {$$ = App2("AM", $1, App1("Build", $3));} */
        | FAIL  {$$ = ATmake("Fail");}
        | SUCC  {$$ = ATmake("Id");}
        | INT strategy {$$ = App2("Path", (ATerm) ATmakeInt($1), $2); }
        | NOT  LPAREN strategy RPAREN {$$ = App1("Not", $3);}
        | WHERE LPAREN strategy RPAREN {$$ = App1("Where", $3);}
        | TEST  LPAREN strategy RPAREN {$$ = App1("Test", $3);}
| LCURLY tvarlist COLON strategy RCURLY
{$$ = App2("Scope", (ATerm) $2, $4);}
| strategy SEMICOLON strategy {$$ = App2("Seq", $1, $3);}
| strategy PLUS strategy  {$$ = App2("Choice", $1, $3);}
```

```
        | strategy LTPLUS strategy    {$$ = App2("LChoice", $1, $3);}
| MU LCID LPAREN strategy RPAREN{$$ = App2("Rec", $2, $4);}
| SOMETOK LPAREN strategy RPAREN{$$ = App1("Some", $3);}
| ONE LPAREN strategy RPAREN {$$ = App1("One", $3);}
| ALL LPAREN strategy RPAREN {$$ = App1("All", $3);}
| THREAD LPAREN strategy RPAREN {$$ = App1("Thread", $3);}
| LPAREN strategylist RPAREN {$$ = tuple_cong($2);}
| LBRACK strategylist
 strategytail RBRACK {$$ = list_cong($2, $3);}
| STRINGTOK {$$ = App1("Match", App1("Str", $1));}
| INT        {$$ = App1("Match",
      App1("Int", (ATerm) ATmakeInt($1)));}
| REAL        {$$ = App1("Match",
      App1("Real", (ATerm) ATmakeReal($1)));}
        ;

strategytail
:  {$$ = ATmake("Call(SVar(\"Nil\"),[])");}
| BAR strategy {$$ = $2;}
;

optstrategylist
:       {$$ = ATmakeList0();}
| LPAREN strategylist RPAREN  {$$ = $2;}
        ;

strategylist
:  {$$ = ATmakeList0();}
| strategy  {$$ = ATmakeList1($1);}
| strategy COMMA strategylist  {$$ = ATinsert($3, $1);}
        ;

/* Strategy definitions */

strategies
:                          {$$ = ATmakeList0();}
| strategies strategy_def  {$$ = ATappend($1, $2);}
        ;

strategy_def
: id optvarlist EQ strategy {$$ = App3("SDef", $1, (ATerm) $2, $4);}
        ;

optvarlist
:  {$$ = ATmakeList0();}
| LPAREN varlist RPAREN  {$$ = $2;}
        ;

varlist :  {$$ = ATmakeList0();}
| id {$$ = ATmakeList1($1);}
| id COMMA varlist  {$$ = ATinsert($3, $1);}
        ;

overlays: {$$ = ATmakeList0();}
| overlay overlays  {$$ = ATinsert($2, $1);}
        ;

overlay : id LPAREN varlist RPAREN EQ term
{$$ = App3("Overlay", $1, (ATerm) $3, $6);}
| id EQ term  {$$ = App3("Overlay", $1, (ATerm) ATmakeList0(), $3);}
        ;

/* Note this allows contexts in overlay definitions; should be excluded
by checking in front-end */
```

PACK-STRATEGO

This module defines a packing algorithm for Stratego.

- command-line option handling

- writing dependencies to .r.dep

- finding file based on path

- parsing the file for a module based on given parser

- flattening

```
module pack-stratego
imports lib pack-graph pack-modules sugar

strategies

  main = pack-modules(pack-stratego, basename)

strategies

  pack-stratego(mkpt) =
        \ root -> (root, (), []) \;
        graph-nodes(Fst; parse(parse-mod, mkpt, !"r"),
                    get-stratego-imports,
                    \ (n,x,xs) -> [x|xs] \ );
        unzip;
        (id, flatten-stratego)

  (* parse :: (filename -> parsetree)
            * (() -> path)
            * (() -> ext)
            -> (filename -> parsetree)
  *)

  parse(parser, mkpath, ext) =
        find-file(mkpath, ext);
        split(id, parser)

  get-stratego-imports =
        \ (_, Specification(xs)) -> xs \;
        filter(\Imports(xs) -> xs\ );
        concat

  flatten-stratego =
        map(\Specification(xs) -> xs\; filter(not(Imports(id))));
        concat;
        \ xs -> Specification(xs) \
rules

  parse-mod : in -> trm
    where <conc-strings> ("pack-stratego",
                          <get-pid; int-to-string>()) => out;
          (* <printnl> (stderr, ["  parsing ", in]); *)
          <call>("parse-mod", ["-silent", "-i", in, "-o", out]);
          <ReadFromFile> out => trm;
          <rm-files> [out]
```

Chapter: Library

The modules in this chapter define common operations on Stratego data types.

LIBRARY FOR STRATEGIES

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

```
module stratlib
imports strategy substitution free-variables


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), [])

strategies

  SVarShape(s) = Call(SVar(s), [])
```

BOUND VARIABLES

The following rules and strategies define which constructs bind variables. The `Bnd` rules define which variables are bound. The `paste` strategies define where new variables should be pasted in case of renaming. The `boundin` strategies define in which arguments of the constructs the variables are binding.

```
rules

  Bind0 : Scope(xs, s) -> xs
  Bind0 : LRule(Rule(t1, t2, s)) -> <tvars> t1
  Bind1 : Let(SDef(f, xs, s1), s2) -> [f]
  Bind2 : SDef(f, xs, s) -> xs
  Bind3 : Rec(x, s) -> [x]

strategies

  tpaste(nwvars) =
        Scope(nwvars, id)

  tboundin(bnd, ubnd, ignore) =
        Scope(ignore, bnd)

  spaste(nwvars) =
          Let(SDef(nwvars; Hd, id, id), id)
        + SDef(id, nwvars, id)
        + Rec(nwvars; Hd, id)

  sboundin(bnd, ubnd, ignore) =
          Let(SDef(ignore, ignore, ubnd), bnd)
        + SDef(ignore, ignore, bnd)
        + Rec(ignore, bnd)
```

FREE VARIABLES AND RENAMING

```
strategies

  tvars = free-vars(Add1, Bind0, tboundin)

  svars = free-vars(Add2, Bind1 + Bind2 + Bind3, sboundin)

  trename = rename(Var, Bind0, tboundin, tpaste)
  srename = rename(SVar, Bind1 + Bind2 + Bind3, sboundin, spaste)

  svars-arity =
    free-vars2(\Call(SVar(f), as) -> [(f, <length> as)]\
              ,(Bind1 + Bind2 + Bind3)
              ,sboundin
              ,{f:?((f,_),f)})

  tsubs = substitute(IsVar)

  ssubs = substitute(IsSVar)

  tsubstitute = substitute(IsVar, Var, Bind0, tboundin, tpaste)

  strename = trename ; srename

  is_var_list = map(Var(id))
  is_svar_list = map(SVar(id))
```

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)))
```

Part II

Compilation

Chapter: 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.

Chapter: Frontend

FRONTEND

```
module frontend
imports normalize-spec
        spec-to-sdefs
        use-def
        check-constructors
strategies

  frontendIO = iowrap(frontend)

  frontend =
        //where(dtime; debug(!"  frontend initialization: "));
        normalize-spec;
        //where(dtime; debug(!"  normalize-spec: "));
        where(spec-use-def);
        //where(dtime; debug(!"  spec-use-def: "));
        ExpandOverlays;
        //where(dtime; debug(!"  ExpandOverlays: "));
        try(CheckConstructors)//;
        //where(dtime; debug(!"  CheckConstructors: "))
```

Chapter: Frontend

NORMALIZE-SPEC

```
module normalize-spec
imports stratego lib
```

The first phase of the front-end is the normalization of specifications. A specification consists of a list of basic specifications (signatures, overlays, rules and strategy definitions) in any order. Normalization collects the basic specifications of each kind and creates a specification of the form

```
  Specification([Signature(id),
                 Overlays(id),
                 Rules(id),
                 Strategies(id)])
```

```
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, [])
  NormBSP : Overlays(ols)     -> ([], ols, [], [])

  Combine : ((ods1, ols1, rdefs1, sdefs1), (ods2, ols2, rdefs2, sdefs2)) ->
            (<conc> (ods1, ods2),
             <conc> (ols1, ols2),
             <conc> (rdefs1, rdefs2),
             <conc> (sdefs1, sdefs2))

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

strategies

  normalize-sigs =
        map(NormBSIG);
        concat

  normalize-specIO = iowrap(normalize-spec)

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

Furthermore, the grammar cannot distighuish term variables from unary constructors. This distinction can only be made based on the signature. Variables are renamed to operator applications by duplicating the specification and mapping one specification to a substitution.

```
rules

  Names   : Signature(bsigs) -> <filter(OpNames); concat> bsigs
  OpNames : Operations(ods)  -> <filter(OpName)> ods
  OpName  : OpDecl(f, ConstType(_)) -> f

  Names   : Overlays(ols) -> <filter(OLName)> ols
  OLName  : Overlay(x, [], t) -> x

strategies

  const-names = filter(Names); concat

  vars-to-consts =
    split(const-names; map(split(id, \x -> Op(x,[])\ ))
         ,id);
    tsubs


rules

  UnaryConstructorName : OpDecl(f, ConstType(_)) -> f
  UnaryConstructorName : Overlay(x, [], t) -> x


rules

  DefLRule : LRule(Rule(t1, t2, s)) ->
             Scope(<tvars> t1, SRule(Rule(t1, t2, s)))

strategies

  define-lrules = topdown(try(DefLRule))
```

USE-DEF

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

In the triple `(u, d, e)` `u` represents the variables used in a build and `d` the variables bound (defined) in a match. `e` represents the variables that are used but never defined. The following strategies define for each construct involved with term variables how it uses or defines variables.

```
strategies

  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) +
         Overlay(id, id, use-term) + Cong(id, map(x))
```

The parallell union of two use-def triples simply takes the point-wise union of each of the sets. In the sequential union the uses of the second triple are are accounted for by the defines in the first triple.

```
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))
```

Because System S expressions can contain choice operators, the use-def information of an expression is represented as a set of use-def triples, one for each path. The strategy `seq-join` joins two sets of triples, where for each pair of triples (`cart` stands for Cartesian product) the sequential union is taken.

```
strategies

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

The `UDjoin` rules define for each construct involved with variables how use-def information is propagated.

```
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 : Cong(f, ss)        -> <seqs-join> ss
  UDjoin : Scope(xs, uds)     -> <map(JoinScope(!xs))> uds
  UDjoin : Overlay(f, xs, t)  -> Overlay(f, xs, <seqs-join> [([],xs,[]),t])

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

For all other constructs the use-def information is combined with `Union`. Finally, the use-def information for an expression is computed by `use-def`.

```
strategies

  default-join =
        explode-term; Snd;
        foldr(![([], [], [])], cart(Union))

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

Each definition of the the specification is checked for valid use of variables. That is, variables should be declared and defined before they are used.

```
strategies

  spec-use-defIO = iowrap(where(spec-use-def))

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

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

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

The following rules transform definitions with use-def information that indicates an error to a readable error message.

```
rules

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

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

  MsgE : [] -> []

  MsgE : [x] ->
         ["variable ", x, ": declared, but not bound"]

  MsgE : [x, y | ys] ->
         ["variables ", [x, y | ys], ": declared, but not bound"]

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

  MkMsg : RDef(l, xs, uds) -> ["error in rule ", l, " : "
                              | <map(MsgR); concat> uds]

  MkMsg : SDef(f, xs, uds) -> ["error in definition ", f, " : "
                              | <map(MsgS); concat> uds]

  MkMsg : Overlay(f, xs, uds) -> ["error in overlay ", f, " : "
                                 | <map(MsgR); concat> uds]

strategies

  err-msg = MkMsg; fatal-error
```

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 sugar stratlib list-sort
```

CONGRUENCES FROM SIGNATURE

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
```

CONSTRUCTORS FROM OPERATORS

```
(*
  MkConsDef : OpDecl(f, FunType(ts, t)) ->
              RDef(f, [], )
              where <map(new)> ts => xs

        RDef(f, [],
           Rule(Op(x1,...,xn), Op(f, xs), id)

  MkConsDef :
    Overlay(f, xs, t) -> RDef(f, [], Op(x1,...,xn), t)
*)
```

CONGRUENCES FROM OVERLAYS

Each overlay defines a congruence operator as well as abstractions to be used in match and build operations.

```
rules

  Overlay-to-Congdef :
    Overlay(f, xs, t) -> SDef(f, xs, <trm-to-cong> t)

  Trm-to-Cong : Var(x)          -> Call(SVar(x), [])
  Trm-to-Cong : Op(f, ts)       -> Call(SVar(f), ts)
  Trm-to-Cong : Str(x)          -> Match(Str(x))
  Trm-to-Cong : Int(x)          -> Match(Int(x))
  Trm-to-Cong : Real(x)         -> Match(Real(x))
  Trm-to-Cong : BuildDefault(x) -> Id

strategies

  trm-to-cong = rec x(try(Op(id, map(x))); Trm-to-Cong)
```

EXPANDING OVERLAYS

```
rules

  ExpOverlay(ols) :
    Op(f, ts) -> <tsubstitute> (sbs, t)
    where ols; fetch(where({xs, t: ?Overlay(f, xs, t);
                                  !(<zip(id)> (xs, ts), t)}
                           => (sbs, t)))

// Note: when overlays in overlay definitions already have been
// expanded the repeat is not necessary. Therefore exp-overlays1 and
// exp-overlays2:

strategies

  exp-overlays1(ols) =
    try(where(not(ols => []));
        topdown(repeat(ExpOverlay(ols))))

  exp-overlays2(ols) =
    try(where(not(ols => []));
        topdown(try(ExpOverlay(ols))))
```

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)))

  DeclareVariables :
        SDef(f, xs, s) -> SDef(f, xs, Scope(<tvars> s, s))
```

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

  RulesToSdefs :
    Specification([
      Signature(bsigs),
      Rules(rdefs),
      Strategies(sdefs)
    ]) ->
    Specification([
      Signature(bsigs),
      Strategies(<concat>
                 [<congdefs> bsigs,
                  <map(RDtoSD)> rdefs,
                  <map(DeclareVariables)> sdefs])
    ])

  ExpandOverlays :
    Specification([
      Signature(bsigs),
      Overlays(ols),
      Rules(rdefs),
      Strategies(sdefs)
    ]) ->
    Specification([
      Signature(bsigs),
      Rules(<exp-overlays2(!ols’)> rdefs),
      Strategies(<conc>(<map(Overlay-to-Congdef)> ols, <exp-overlays2(!ols')> sdefs))
    ])
    where <exp-overlays1(!ols)> ols => ols'

strategies

  ExpandOverlaysIO = iowrap(ExpandOverlays)

  spec-to-sdefs =
        Spec-to-Sdefs;
        strename
```

DESUGARING

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

```
module desugar
imports sugar stratlib list-sort use-def lib
```

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(!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
```

TERM EXPLOSION AN CONSTRUCTION

```
rules

  Expl  : Match(t[Explode(t1, t2)]) ->
          Scope([x],
                Seq(Match(t[Var(x)]),
                    Where(BAM(Prim("_ST_explode_term"),
                          Var(x),
                          Op("TCons", [t1, Op("TCons", [t2, Op("TNil",[])])])
                          ))))
          where new => x

  Expl  : Build(t[Explode(t1, t2)]) ->
          Scope([x],
                Seq(BAM(Prim("_ST_mkterm"),
                        Op("TCons", [t1, Op("TCons", [t2, Op("TNil",[])])]),
                        Var(x)),
                    Build(t[Var(x)])))
          where new => x

  Expl : ExplodeCong(s1, s2) ->
         Seq(Prim("_ST_explode_term"),
         Seq(Cong("TCons", [s1, Cong("TCons", [s2, Cong("TNil",[])])]),
             Prim("_ST_mkterm")))
```

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', Call(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', Call(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', Call(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) + Expl))

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

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

NEEDED DEFINITIONS

Extract those definitions that are needed for the main strategy and join the bodies of operators with multiple definitions.

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

JOINING DEFINITIONS

```
strategies

  joindefs = JoinDefs1 <+ JoinDefs2

rules

  JoinDefs1 : [sdef] -> sdef

  JoinDefs2 : defs @ [SDef(f, xs, s) | _] -> SDef(f, ys, Choices(ss))
        where ?ys  <= <map(new)> xs;
              ?ys' <= <map(MkCall)> ys;
              ?ss  <= <map({zs, s: ?SDef(f, zs, s); !<ssubs> (zs, ys', s)})>
                         defs
```

OBTAINING NEEDED DEFINITIONS

```
strategies

  needed-defs =
    \ defs -> (("main", 0), defs, []) \;
    graph-nodes-undef(get-definition
                     ,svars-arity
                     ,\ (_,x,d) -> [x|d] \ );
    (NoMissingDefs <+ MissingDefs; <exit> 1)

  get-definition =
       CongruenceDef//; debug
    <+ OverloadedDef; joindefs
    <+ NonOverloadedDef; joindefs
```

A strategy operator f with arity n is needed. All definitions for the operator are fetched and joined. Note that this entails that (1) operators can be overloaded and (2) there can be more than one definition of an operator.

```
rules

  OverloadedDef :
    ((f, n), defs) -> fdefs
    where <filter(SDef(?f,where(length => n),id))> defs => fdefs

  NonOverloadedDef :
    ((f, 0), defs) -> fdefs
    where <filter(SDef(?f,id,id))> defs => fdefs

  CongruenceDef :
    ((Mod(c, mod), n), defs) -> fdef
    where <DefineCongruence> (c, mod, n) => fdef

  NoMissingDefs :
    (defs, []) -> defs

  MissingDefs :
    (defs, [f|fs]) -> defs
    where <map(MissingDefMod <+ MissingDef)> [f|fs]

  MissingDef :
    (f, n) -> <error> ["error: operator ", f, "/", n, " undefined "]

  MissingDefMod :
    (Mod(c, m), n) -> <error> ["error: operator ", c, "^", m , "/", n, " undefined "]
```

DISTRIBUTING CONGRUENCES

For each constructor c, there is a corresponding distributing congruence c^D, defined according to the following scheme:

```
  c^D(s1,...,sn) : Pair(c(x1,...,xn),env) -> c(y1,...,yn)
  where <s1> Pair(x1,env) => y1;
        ...
        <sn> Pair(xn, envn) => yn
```

This is implemented by the following rules.

```
overlays
  OpPair(t1, t2) = Op("Pair", [t1,t2])

rules
  DefineCongruence :
    (c, "D", n) ->
    SDef(Mod(c, "D"), ss, Scope([env | <conc>(xs1, ys1)],
                                SRule(Rule(OpPair(Op(c, xs2), Var(env)),
                                           Op(c, ys2),
                                           Seqs(conds)))))
    where ?env <= new;
```

```
        ?(conds, ss, xs1, xs2, ys1, ys2)
           <= <copy(MkDistApplication); tuple-unzip(id)> (n, Var(env))

  MkDistApplication :
    env -> (BAM(Call(SVar(s),[]), OpPair(Var(x), env), Var(y)),
            s, x, Var(x), y, Var(y))
    where new => s; new => x; new => y
```

THREADING CONGRUENCES

For each constructor `c`, there is a corresponding threading congruence `c^T`, defined according to the following scheme:

```
  c^T(s1,...,sn) : Pair(c(x1,...,xn),e-first) -> Pair(c(y1,...,yn), e-last)
  where <s1> Pair(x1,e-first) => Pair(y1,e2);
              ...;
        <sn> Pair(xn, en) => Pair(yn, e-last)
```

The following rules implement this scheme:

```
  DefineCongruence :
    (c, "T", n) ->
    SDef(Mod(c, "T"), ss, Scope([e-first | <concat> [es, xs1, ys1]],
                                SRule(Rule(OpPair(Op(c, xs2), Var(e-first)),
                                           OpPair(Op(c, ys2), Var(e-last)),
                                           Seqs(as)))))
    where ?[e-first | es] <= <copy(new)> (<add>(n,1), ());
          ?e-last <= <last> es;
          ?(as, ss, xs1, xs2, ys1, ys2)
          <= <zipr(MkThreadApplication); tuple-unzip(id)> ([e-first | es], es)

  MkThreadApplication :
    (e1,e2) -> (BAM(Call(SVar(s),[]), OpPair(Var(x), Var(e1)),
                                      OpPair(Var(y), Var(e2))),
                s, x, Var(x), y, Var(y))
    where new => s; new => x; new => y
```

INLINING

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

```
module inlining
imports strategy stratlib lib

rules

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

  Inl2 : Let(SDef(f, [x| xs], s1), s2[Call(SVar(f), ss)]) ->
         Let(SDef(f, [x| xs], s1),
             s2[<ssubs ; strename> ([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(!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 <length> ss => n;
              <fetch(SDef(?f, where(length => n); ?xs , ?s))> env

  ExpandCall :
        (Call(SVar(f), ss), env) -> (Call(SVar(g), ss), env)
        where <fetch(?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, 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

  inlineIO = iowrap(inline)

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, <ssubs> (<mksbs>(), s))
  where (mktbl; fetch(?(x, y))) <+ (!x => y)
```

OPTIMIZER

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

strategies

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

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
  I14 : CongWld(ss)     -> Id where <map(?Id)> ss
  I15 : App(Id, t)      -> t
  I16 : Match(Wld)      -> Id

  I13 : Cong(f, ss)     -> MatchFunA(Fun(f, <length> ss), [], [], Id)
        where <map(?Id)> ss
strategies

  ElimId = 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(?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)))
```

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

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

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

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

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

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

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


  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')] -> Rec(x, s[Call(x,[])])
```

Distribution and recursion

```
        (* 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)
```

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))
```

OPTIMIZATION

```
module optimization
imports strategy list simplification distribution 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> ([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) -> App(s, t) \ )> (ss, ts))),
            s[Id](firstInSeq))

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

  BuildCong :
        Seq(Build(Op(f, ts)), Seq(MatchFun(f, n), s[CongWld(ss)])) ->
        Seq(Build(Op(f, <zip(\ (s, t) -> App(s, t) \ )> (ss, ts))),
            s[Id](firstInSeq))
        where <length> ts => n

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

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

Strategies for application of simplification rules.

```
strategies

  simplify = ElimId + F + Ass + P + FuseScope + D' + ChoiceMerge (* + M *)

  fuse = ElimId + Ass + CongBuild + BuildCong + AppCong

  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))
```

AUTOMATON

Matching automaton instructions.

```
module automaton
imports terms lib
signature
  sorts Aut
  constructors
    Down      : Int * Aut       -> Aut
    Up        : Aut             -> Aut
    Accept    : List(Int)       -> Aut
    Case      : List(Aut) * Aut -> Aut

    MatchVars : List(String) * Path * Aut -> Aut
    MatchFunA  : String  * List(String) * Path * Aut -> Aut

    MatchTerm : Path * Term * Aut -> Aut
```

Generation of a matching automaton from a term.

```
rules

  Aut1 : MatchTerm(p, Wld, c)    -> MatchVars([], p, c)

  Aut4 : MatchTerm(p, Var(x), c) -> MatchVars([x], p, c)

  Aut4a : MatchTerm(p, As(Var(x), t), c) ->
          MatchVars([x], p, MatchTerm(p, t, c))

  Aut2 : MatchTerm(p, Str(x), c) -> MatchFunA(Str(x), [], p, c)

  Aut3 : MatchTerm(p, Int(x), c) -> MatchFunA(Int(x), [], p, c)

  Aut3 : MatchTerm(p, Real(x), c) -> MatchFunA(Real(x), [], p, c)

  Aut5 : MatchTerm(p, Op(f, ts), c) ->
         MatchFunA(Fun(f, <length> ts), [], p, MatchKids(0, p, ts, c))

  Aut6 : MatchKids(n, p, Nil, c) -> c

  Aut7 : MatchKids(n, p, Cons(t, ts), c) ->
         Down(n, MatchTerm(Cons(n, p), t,
                 Up(MatchKids(<plus> (n, 1), p, ts, c))))

  Aut8 : MatchTerm(p, BuildDefault(t), c) ->
         MatchVars([], p, c)

  AutInit : Pat(t, s) -> MatchTerm([], t, Accept(s))
```

```
strategies

  pat-to-aut = AutInit;
               topdown(repeat(Aut1 + Aut2 + Aut3 + Aut4a +
                              Aut4 + Aut5 + Aut6 + Aut7 + Aut8))

  mk-automata = map(pat-to-aut)
```

Optimization of automata

```
rules

  O : Down(n, Up(c)) -> c
  O : MatchVars([], p, c) -> c

strategies

  opt-automaton = downup(repeat(O))
```

Merging of automata for multiple patterns

merging traversal operations

```
rules

  Mrg : Merge(Down(n, c1), Down(n, c2)) -> Down(n, Merge(c1, c2))

  Mrg : Merge(Up(c1), Up(c2)) -> Up(Merge(c1, c2))

  Mrg : Merge(Accept(s1), Accept(s2)) -> Accept(Choice(s1, s2))
```

skipping up

When a pattern does not check some subtree this will be indicated by an `Up` where other pattersn that do inspect the subtree have some other operation. The `Up` is pushed inside until it meets its matching `Up` in the other pattern.

```
rules

  Mrg : Merge(Down(n, c1), Up(c2)) ->
        Down(n, Merge(c1, Up(Up(c2))))

  Mrg : Merge(Up(c1), Down(n, c2)) ->
        Down(n, Merge(Up(Up(c1)), c2))

  Mrg : Merge(MatchFunA(f, xs, p, c1), Up(c2)) ->
```

```
       MatchFunA(f, xs, p, Merge(c1, Up(c2)))

  Mrg : Merge(Up(c1), MatchFunA(f, xs, p, c2)) ->
       MatchFunA(f, xs, p, Merge(Up(c1), c2))

  Mrg : Merge(MatchVars(xs, p, c1), Up(c2)) ->
        MatchVars(xs, p, Merge(c1, Up(c2)))

  Mrg : Merge(Up(c1), MatchVars(xs, p, c2)) ->
        MatchVars(xs, p, Merge(Up(c1), c2))

  Mrg : Merge(Up(c), Case(cs)) ->
        Case(<map({c’: ?c’; !Merge(Up(c), c’)})> cs)

  Mrg : Merge(Case(cs), Up(c)) ->
        Case(<map({c’: ?c’; !Merge(Up(c), c’)})> cs)
```

Merging with Accept

```
rules

  Mrg : Merge(Up(c1), Accept(c2)) ->
       Up(Merge(c1, Accept(c2)))

  Mrg : Merge(Accept(c1), Up(c2)) ->
       Up(Merge(Accept(c1), c2))

  Mrg : Merge(Down(n, c1), Accept(c2)) ->
       Down(n, Merge(c1, Up(Accept(c2))))

  Mrg : Merge(Accept(c1), Down(n, c2)) ->
       Down(n, Merge(Up(Accept(c1)), c2))

  Mrg : Merge(MatchFunA(f, xs, p, c1), Accept(c2)) ->
       MatchFunA(f, xs, p, Merge(c1, Accept(c2)))

  Mrg : Merge(Accept(c1), MatchFunA(f, xs, p, c2)) ->
       MatchFunA(f, xs, p, Merge(Accept(c1), c2))

  Mrg : Merge(MatchVars(xs, p, c1), Accept(c2)) ->
        MatchVars(xs, p, Merge(c1, Accept(c2)))

  Mrg : Merge(Accept(c1), MatchVars(xs, p, c2)) ->
        MatchVars(xs, p, Merge(Accept(c1), c2))

  Mrg : Merge(Accept(c), Case(cs)) ->
        Case(<map({c’: ?c’; !Merge(Accept(c), c’)})> cs)

  Mrg : Merge(Case(cs), Accept(c)) ->
```

```
        Case(<map({c': ?c'; !Merge(Accept(c), c')})> cs)
```

MERGING MATCHING OPERATIONS

```
rules

  Mrg : Merge(MatchVars(xs, p, c1), MatchVars(ys, p, c2)) ->
       MatchVars(<conc> (xs, ys), p, Merge(c1, c2))

  Mrg : Merge(MatchFunA(f, xs, p, c1), MatchFunA(f, ys, p, c2)) ->
       MatchFunA(f, <union> (xs, ys), p, Merge(c1, c2))

  Mrg : Merge(MatchFunA(f, xs, p, c1), MatchFunA(g, ys, p, c2)) ->
       Case([MatchFunA(f, xs, p, c1), MatchFunA(g, ys, p, c2)])
       where <not(eq)> (f, g)

  Mrg : Merge(MatchFunA(f, ys, p, c1), MatchVars(xs, p, c2)) ->
       Case([MatchFunA(f, <union> (xs, ys), p, Merge(c1, c2)),
             MatchVars(xs, p, c2)])

  Mrg : Merge(MatchVars(xs, p, c1), MatchFunA(f, ys, p, c2)) ->
       Case([MatchFunA(f, <union> (xs, ys), p, Merge(c1, c2)),
             MatchVars(xs, p, c1)])

  Mrg : Merge2(MatchVars(xs, p, c1), MatchFunA(f, ys, p, c2)) ->
        MatchFunA(f, <union> (xs, ys), p, Merge(c1, c2))

  Mrg : Merge2(MatchVars(xs, p, c1), MatchVars(ys, p, c2)) ->
        MatchVars(<conc> (xs, ys), p, Merge(c1, c2))
```

MERGING WITH CASE

```
rules

  MrgInsert1 :
        Merge(MatchFunA(f, xs, p, c1), Case(cs)) -> Case(cs')
        where <fetch({ys, c2: ?MatchFunA(f, ys, p, c2);
                      !MatchFunA(f, <union> (xs, ys), p, Merge(c1, c2))})>
              cs => cs'

  MrgInsert2 :
        Merge(MatchFunA(f, xs, p, c1), Case(cs)) ->
        Case(Cons(MatchFunA(f, <union> (xs, ys), p, Merge(c1, c2)), cs))
        where <fetch(?MatchVars(ys, p, c2))> cs

  MrgInsert3 :
        Merge(MatchFunA(f, xs, p, c1), Case(cs)) ->
        Case(Cons(MatchFunA(f, xs, p, c1), cs))
  Mrg :
        Merge(MatchVars(xs, p, c1), Case(cs)) -> Case(cs')
        where
         <map({c2:?c2; !Merge2(MatchVars(xs, p, c1), c2)})> cs => cs'';
         ((<fetch(MatchVars(id,id,id))> cs; !cs'')
           <+ !Cons(MatchVars(xs, p, c1), cs'')) => cs'


rules

  mk-merge : (a, b) -> Merge(a, b)

  C2L : Choice(s1, s2) -> <conc> (s1, s2)
strategies

  merge = mk-merge;
          topdown(repeat(Mrg <+ (MrgInsert1 <+ MrgInsert2 <+ MrgInsert3)))

  merge-automata =
         foldr(!Accept(Fail), merge)

  choices-to-list = choicemap({x: ?x; ![x]});
                    choicebu(C2L)

  mk-automaton =
        choices-to-list;
        mk-automata;
        merge-automata
```

MATCHING-TREE

```
module matching-tree
imports strategy stratlib optimization instructions fixpoint-traversal
        automaton lib
signature
  constructors
    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(!s2) <+
                     Choice(x, x) <+
                     AddVarPart'(!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, <length> ts), <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) +
                                  Real(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)

  match-tree = mk-automaton
```

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), Pat(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')]))

rules

  MakeLinear' :
        Scope(xs, Seq(Match(t[Cons(t'[Var(x)],ts'[Var(x)])], s))) ->
        Scope(Cons(y, xs),
              Seq(Match(t[Cons(t',ts'[Var(y)])]),
              Seq(Build(Var(y)), Seq(Match(Var(x)), s))))
        where new => y

  MakeLinear :
    Scope(xs, Seq(Match(t), s)) ->
    Scope(<conc> (xs, ys), Seq(Match(t'), Seq(ts, s)))
    where <rec x((V1 + V2) <+ thread(x))> Pair(t, ([], [], Id))
                                       => Pair(t', (xs, ys, ts))

  V1 : Pair(Var(x), (xs, ys, ts)) ->
       Pair(Var(x), (Cons(x, xs), ys, ts))
       where <not(in)> (x, xs)

  V2 : Pair(Var(x), (xs, ys, ts)) ->
       Pair(Var(x), (xs, Cons(y, ys),
                     Seq(Build(Var(x)), Seq(Match(Var(y)), ts))))
       where <in> (x, xs); new => y

  mk-aut : Match(t) -> <pat-to-aut> Pat(t, Id)


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(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

  mk-match-tree =
        is-scoped-case;
        make-linear;
```

```
      lift-scopes;
      Scope(id, lift-continuations;
                rec x(Let(id, x) <+ match-tree))

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

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

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

Chapter: Backend
Section: Abstract Machine

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

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, n), Cons(TravInit, Cons(AllBuild, is))) ->
       Cons(MatchFun(f, n), 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)

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

Chapter: Backend
Section: Compilation

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

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

Chapter: Backend
Section: Compilation

BACKEND

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

  main = iowrap(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(Cspecial <+ C)));
            flatten-blocks;
            peephole;
            Assemble
```

Make a strategy into an instruction

```
rules

  MkInstr : s -> Instr(s, [], 0)

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

Chapter: Backend
Section: Compilation

COMPILER

```
module compiler
imports instructions strategy list substitution automaton
```

semi-instructions: strategies embedded into instructions

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

Looking up variables in the environment

```
rules

  io-index : (x, env) -> (i, o)
     where <fetch({i, xs: ?(i, xs); !(i, <get-index> (x, xs))}; ?(i, o))> env
```

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
```

Chapter: Backend
Section: Compilation

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)
```

Matching automaton

```
  C : Instr(Down(n, s), env, rcs) ->
      Block([Arg(n),
             Instr(s, env, rcs)])

  C : Instr(Up(s), env, rcs) ->
      Block([Tpop,
             Instr(s, env, rcs)])

  C : Instr(Accept(s), env, rcs) ->
      Instr(s, env, rcs)

  C : Instr(MatchVars([], p, s), env, rcs) ->
      Instr(s, env, rcs)

  C : Instr(MatchVars(Cons(x, xs), p, s), env, rcs) ->
      Block([ins,
             Instr(MatchVars(xs, p, s), env, rcs)])
      where <io-index> (x, env) => (i, o);
            ((<eq> (i, rcs); !MatchVar(o)) <+ !MatchVard(i, o) => ins)

  C : Instr(MatchFunA(Fun(f,n), xs, p, s), env, rcs) ->
      Block([MatchFun(f,n),
             Instr(MatchVars(xs, p, s), env, rcs)])
  C : Instr(MatchFunA(Str(x), xs, p, s), env, rcs) ->
      Block([MatchString(x),
             Instr(MatchVars(xs, p, s), env, rcs)])

  C : Instr(MatchFunA(Int(x), xs, p, s), env, rcs) ->
      Block([MatchInt(x),
             Instr(MatchVars(xs, p, s), env, rcs)])

  C : Instr(MatchFunA(Real(x), xs, p, s), env, rcs) ->
      Block([MatchReal(x),
             Instr(MatchVars(xs, p, s), env, rcs)])

  C : Instr(Case(cs), env, rcs) ->
      Block([Instr(Cases(cs', sc), env, rcs),
             Instr(s, env, rcs),
             Label(sc)])
      where new => sc;
            (<at_suffix({cs: Cons(MatchVars(id,id,id);?s, ?cs); !cs})> cs
            <+ (!Fail => s; !cs)) => cs'

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

  C : Instr(Cases(Cons(MatchFunA(f, xs, p, s), cases), sc), env, rcs) ->
      Block([<CasePrefix> (f, fc),
             Instr(MatchVars(xs, p, s), env, rcs),
             Goto(sc),
             Label(fc),
             Instr(Cases(cases, sc), env, rcs)])
      where new => fc

  CasePrefix : (Fun(f, n), fc) -> MatchFunFC(f, n, fc)
  CasePrefix : (Int(n), fc)  -> MatchIntFC(n, fc)
  CasePrefix : (Real(n), fc) -> MatchRealFC(n, fc)
  CasePrefix : (Str(x), fc)  -> MatchStringFC(x, fc)
```

Cases (Old 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(Real(n)), fc) -> MatchRealFC(n, fc)
  CasePrefix' : (Match(Str(x)), fc)  -> MatchStringFC(x, fc)
```

Cases (Old 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, <length> ss),
             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)])
```

Chapter: Backend
Section: Compilation

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
```

Thread

```
  C : Instr(Thread(s), env, rcs) ->
      Block([ThreadInit(),
             Label(c1),
             ThreadNextSon(c2),
             Instr(s, env, rcs),
             ThreadSetEnv(),
             Goto(c1),
             Label(c2),
             ThreadBuild()])
      where new => c1 ; new => c2
```

Scope

```
  C : Instr(Scope(xs, s), env, i) ->
      Block([Epushd(i, o),
             Instr(s, Cons((i, xs), env), <add> (i, 1)),
             Epopd(i, o)])
      where <length> xs => o
```

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) -> ins
      where <io-index> (x, env) => (i, o);
            ((<eq> (i, rcs); !MatchVar(o)) <+ !MatchVard(i, o) => ins)

  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)

  C’ : Instr(Match(Real(n)), env, rcs) ->
      MatchReal(n)
```

Building Terms

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

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

  C : Instr(Build(Real(x)), env, rcs) ->
      BuildReal(x)

  C : Instr(Build(Var(x)), env, rcs) -> ins
      where <io-index> (x, env) => (i, o);
            ((<eq> (i, rcs); !BuildVar(o)) <+ !BuildVard(i, o) => ins)

  C : Instr(Build(BuildDefault(t)), env, rcs) ->
      Instr(Build(t), env, rcs)

  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])
```

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 topmost. 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

 Cspecial :: 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), []))))}

  Cspecial = 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"))))}

  Cspecial = CsFetch

rules

  CsFetch ::
    Instr(Fetch(?s; IsMatch), ?env, ?rcs) -->
    ! Block([
        Tdupl,
        Label(a),
        MatchFun("Cons",2),
        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.

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.

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'

strategies

  main = iowrap(Main)
```

Chapter: Other Operations
Section: Experiments

Consider as an example of optimization:

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

Chapter: Compiler

In this chapter we put the compiler components developed in the previous chapters together into the Stratego Compiler *sc*.

Chapter: Compiler
Section: Glue

sc: STRATEGO COMPILER

```
module sc
imports lib sugar
signature
  sorts Option
  constructors
    Dir      : String -> Option
    ExecDir  : String -> Option
    InclDir  : String -> Option
    CInclDir : String -> Option
    CLibDir  : String -> Option
    Input    : String -> Option
    Main     : String -> Option
    AST      : Option
    Ignore   : Option
    CC       : Option
    NORM     : Option
```

Processing the command-line options

```
strategies

  main = sc

  sc = (process-sc-options <+ sc-usage; <exit> 1);
       ((need-help(sc-usage; <exit> 1), id)
        <+ sc-announce;
           parse;
           output-ast;
           add-main;
           core;
           cc1;
           cc2;
           try(not((option-defined(?NORM()), id));
               (id, remove-intermediates));
           <printnl>(stderr, ["compilation succeeded"]);
           <exit> 0
        <+ <printnl>(stderr, ["compilation failed"]);
           <exit> 1
       )

  process-sc-options =
        where(filter-options(?"-I") => incl);
        where(filter-option-args(?"-CI") => cincl);
        where(filter-option-args(?"-CL") => clib);
        parse-options(sc-options <+ io-options);
        (option-defined(Input(?in));
        \ opts ->
          ([InclDir(incl), CInclDir(cincl), CLibDir(clib) | opts],
           (in, ".r")) \
        <+ \opts -> ([Help | opts], "")\ )

  sc-options =
        ArgOption("-e",          \x -> ExecDir(x)\ )
        + ArgOption("-I",        !Ignore) // \x -> InclDir(x)\ )
        + ArgOption("--Include", !Ignore) // \x -> InclDir(x)\ )
        + ArgOption("-CI",       !Ignore) // \ (x,y) -> CInclDir(x)\ )
        + ArgOption("-CL",       !Ignore) // \ (x,y) -> CLibDir(x)\ )
        + Option("-CC",          !CC )
        + Option("-norm",        !NORM )
        + ArgOption("-i",        \x -> Input(<basename> x)\ )
        + ArgOption("--main",    \x -> Main(x)\ )
        + ArgOption("-m",        \x -> Main(x)\ )
        + Option("--ast",        !AST )

strategies

  sc-usage =
  sc-version;
  <printnl>(stderr,
            ["Usage: sc [options] -i file\n",
             "Options:\n",
             "  -i spec     Compile specification spec\n",
             "  -o target   Name executable target\n",
             "  --main s    Name main strategy [default: main]\n",
             "  -I dir      Look in dir for imported Stratego modules\n",
             "  -CI dir     Look in dir for C include files\n",
             "  -CL dir     Look in dir for C object libraries\n",
             "  --ast       Output abstract syntax of specification\n",
             "  -h|--help   Display this message"
            ])

  sc-version =
  (option-defined(DeclVersion(?version)) <+ !"" => version);
  where(<printnl>(stderr, ["sc version ", version]))

  sc-announce =
  try((option-defined(Verbose), id);
      (sc-version, id))
```

Parsing specifications

```
strategies

  parse =
    (option-defined(InclDir(?incl));
     option-defined(ExecDir(?edir));
     try(option-defined(Output(?out))), id);
    (id, pipe'(<pref(!edir)> "/pack-stratego", !".tree",
               <conc> (<!["-dep", out] <+ ![]>(), ["--silent" | incl])))
```

```
strategies

  output-ast =
    try((option-defined(AST), (?file, ?ext));
        <printnl>(stderr, ["abstract syntax written to ", file, ext]);
        <exit> 0)
```

Adding main strategy

(*** What happens if the specification already contains a main strategy? ***)

```
rules

  AddMain(m) :
    Specification(sects) ->
    Specification([Strategies([SDef("main", [], Call(SVar(<m>()), []))])
                  | sects])

strategies

  add-main =
    ((option-defined(Main(?m)), id);
     (id, transform-file(AddMain(!m), !".tree1"))
     <+ (id, transform-file(id, !".tree1")))
```

The core of the compiler consists of the components that transform a specification to abstract machine instructions.

```
strategies

  core =
    (list(try(ExecDir(?dir))), id);
    (id, frontend(!dir);
         extract(!dir);
         inline(!dir);
         optimizer(!dir);
         matching-tree(!dir);
         optimizer(!dir);
         backend(!dir);
         postprocess(!dir);
         pp-instructions(!dir)
    )

rules

  pref(d) : x -> <conc-strings> (<d>(), x)

strategies

  frontend(d)        = pipe(<pref(d)> "/frontend",      !".s1")
  extract(d)         = pipe(<pref(d)> "/extract",       !".s2")
  inline(d)          = pipe(<pref(d)> "/inline",        !".s")
  optimizer(d)       = pipe(<pref(d)> "/optimizer",     !".so1")
  matching-tree(d)   = pipe(<pref(d)> "/matching-tree", !".so2")
  backend(d)         = pipe(<pref(d)> "/backend",       !".i1")
  postprocess(d)     = pipe(<pref(d)> "/postprocess",   !".i")
  pp-instructions(d) = pipe(!"pp-instructions",         !".c")
          (* pipe(<pref(d)> "pp-instructions",    !".c")/*)


strategies

rules

  I-option : x -> <conc-strings>("-I", x)
  L-option : x -> <conc-strings>("-L", x)

strategies

  lib(d)      = <conc-strings>(<d>(), "/lib")
  liblib(d)   = <conc-strings>(<lib(d)>(), "/lib")
  include(d)  = <conc-strings>(<lib(d)>(), "/include")
  libstrat(d) = <conc-strings>(<lib(d)>(), "/stratego")

  gcc = \ args -> <call>("gcc", args) \

  cc1 =
        where(<printnl>(stderr, ["compiling"]));
        (list(try(CInclDir(?cincl))), id);
        (id, where(conc-strings => cfile);
             (id, !".o");
             where(conc-strings => target);
             where(<gcc> <conc> (cincl,["-c", cfile,"-o", target])))

  cc2 =
        where(<printnl>(stderr, ["linking"]));
        (list(try(Dir(?dir) + CLibDir(?clib) + Output(?out))), id);
        (id, where(conc-strings => ofile);
             (try(!out), !"");
             where(conc-strings => target);
             where(<gcc> <conc>([ofile, "-o", target],
                                <map(split-at-space); concat> clib)))

  remove-intermediates =
    ?(base, _);
    where(<rzip(conc-strings); rm-files>
            (base, [".tree", ".tree1", ".s", ".s1", ".s2",
                    ".so1", ".so2", ".i1", ".i", ".o"]))
```

Part III

Run-Time System

DISPLAYS VS STATIC LINKS

Dealing with static scope requires keeping track of the currently enclosing scope. In simple cases it is sufficient to take an offset from the top of the environment stack.

```
environment[esp - o]
```

But when several activations of the same scope can be created due to recursion, this mechanism fails because the static offsets to variables in outer scopes do not correspond to the dynamic offsets on the actual stack.

Common solutions for this problem are static links, displays and lambda lifting (e.g., [1]).

Displays use a global array that give for each nesting depth the most recent activation record for that nesting depth. This gives fast access to the stack. However, it turns out that this mechanism requires a lot of bookkeeping and it does not combine well with backtracking. For each entry in the display a stack of previous activation records has to be maintained.

A static link is a reference to the activation record of the previous nesting level. Finding a stack offset using static links requires following the links until the activation record of the right level is found.

The Stratego run-time system uses static links. There is a single stack that keeps track of static links. A static link consists of a pointer to the environment stack, the nesting depth and a pointer to the static link of the enclosing activation record.

Chapter: Run-time System
Section: stratego.h

```
/*

Copyright (C) 1998, 1999 Eelco Visser <visser@acm.org>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

*/

/* $Id: stratego.h,v 1.2 2000/01/19 16:24:01 visser Exp $ */

#include <aterm2.h>
#include "aterm-extension.h"
#include "debug.h"
#include "util.h"
#include "options.h"
#include "svm.h"
```

Chapter: Run-time System
Section: aterm-extension.h

```
/*

Copyright (C) 1998, 1999 Eelco Visser <visser@acm.org>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

*/

/* $Id: aterm-extension.h,v 1.3 2000/02/19 12:23:28 visser Exp $ */

#include <aterm2.h>

/* Extension of ATerm library */

#define t_string(t) ATgetName(ATgetSymbol(t))

#define t_is_appl(t) (ATgetType(t) == AT_APPL)

#define t_is_string(t) (t_is_appl(t) && ATisQuoted(ATgetSymbol(t)))

#define ATisReal(t) (ATgetType(t) == AT_REAL)
#define ATisInt(t)  (ATgetType(t) == AT_INT)

ATerm list_to_consnil(ATerm t);
ATerm list_to_tconstnil(ATerm t);
ATerm list_to_consnil_op(ATermList t);
ATerm list_to_consnil_op_tl(ATermList t, ATerm tl);
ATerm list_to_consnil_shallow(ATerm t);
ATerm list_to_tconstnil_op(ATermList t);
ATerm consnil_to_list(ATerm t);
ATerm consnil_to_list_shallow(ATerm t);
ATerm tuple_cong(ATermList t);
ATerm list_cong(ATermList t, ATerm tl);

ATerm ATmakeString(char *name);
ATerm ATmakeStringQ(char *name);
ATbool ATisString(ATerm t);
```

Chapter: Run-time System
Section: aterm-extension.h

```
ATbool ATisThisString(ATerm t, char *name);
ATermList ATmap(ATermList l, ATerm (* f)(ATerm));
ATbool AThasName(ATerm t, char *name);

ATerm App0(char *name);
ATerm App1(char *name, ATerm arg1);
ATerm App2(char *name, ATerm arg1, ATerm arg2);
ATerm App3(char *name, ATerm arg1, ATerm arg2, ATerm arg3);
ATerm App4(char *name, ATerm arg1, ATerm arg2, ATerm arg3, ATerm arg4);

ATerm AppN(char *name, ATermList args);

#define ATisInt(t) (ATgetType(t) == AT_INT)
```

Chapter: Run-time System
Section: debug.h

```
/*

Copyright (C) 1998, 1999 Eelco Visser <visser@acm.org>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

*/

/* Booleans */
#include <assert.h>

#define true 1
#define false 0

extern int debugging;

/* #define DEBUG */

#define toe(t) (ATisEmpty(t) ? t : ATgetFirst(t))

#ifdef DEBUG
#define trace(x) Trace("", x, term_stack, return_stack, environment, \
       choice_stack, lchoice_stack)
#define debug(x) Trace(x, -1, term_stack, return_stack, environment, \
       choice_stack, lchoice_stack)
#define debugs(x) if (debugging > 1) x
#else
#define trace(x)
#define debug(x)
#define debugs(x)
#endif
```

Chapter: Run-time System
Section: util.h

```
/*

Copyright (C) 1998, 1999 Eelco Visser <visser@acm.org>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

*/

#ifndef UTIL_H
#define UTIL_H

#include <string.h>

#ifndef streq
#  define streq(s,t) (!(strcmp(s,t)))
#endif

#ifndef MIN
#  define MIN(a,b) ((a) < (b) ? (a) : (b))
#endif

#ifndef MAX
#  define MAX(a,b) ((a) > (b) ? (a) : (b))
#endif

#define IDX_TOTAL              0
#define IDX_MIN                1
#define IDX_MAX                2

#define STATS(array, value)  \
  array[IDX_TOTAL] += value; \
  if(value < array[IDX_MIN]) \
    array[IDX_MIN] = value;  \
  if(value > array[IDX_MAX]) \
    array[IDX_MAX] = value
```

```
#define MYMAXINT 0x7FFFFFFF

#endif
```

Chapter: Run-time System
Section: options.h

```
/*

Copyright (C) 1998, 1999 Eelco Visser <visser@acm.org>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

*/

#include <aterm2.h>

ATbool silent;
char *program_name;
char *input_file;
char *output_file;
ATbool binary_output;
int  show_stats;
ATermList includes;

void process_options(int argc, char *argv[]);
```

Chapter: Run-time System
Section: svm.h

```
/*

Copyright (C) 1998, 1999 Eelco Visser <visser@acm.org>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

*/

/*

$Id: svm.h,v 1.7 2000/06/12 14:04:53 visser Exp $

Implementation of abstract machine instructions for strategy core language

TODO

* The environment is not saved before a choice. This means that some
  variables can become instantiated in a failing alternative and then
  not be restored to unitialized.

* Make stack sizes dynamic or at least configurable from the
  command-line.

*/

#ifndef __defined_stratego_svm_h
#define __defined_stratego_svm_h

/* Constants for stack-size */

#define TSIZE 15000
#define RSIZE 15000
#define ESIZE 15000
#define DSIZE 10000
#define CSIZE 15000
#define NR_COUNTERS 5000
#define NR_RULECOUNTERS 5000
```

```
/* Idiom */

#define between(a, x, b) (assert((a) <= (x)), assert((x) <= (b)))

/* Error handling */

#define panic(s) ATfprintf(stderr, "fatal error: %s\n", s); exit(1);

/* Dispatching */

#define GO(s) {register void *a; if((a = s) != NULL) goto *a;}

/* Profiling */

extern int rule_counter;
extern int match_counter;
extern int build_counter;

struct {
  char *name;
  long int count;
} rule_counters[NR_RULECOUNTERS];

long int cur_rule_counter;

#define CountRule(f) rule_counters[f].count++; rule_counter++;

#define RuleCounter(s1, s2, n) \
   rule_counters[n].name = s2; rule_counters[n].count = 0;


/* Counters */

struct {
  int count;
  int ok;
} counter_stack[NR_COUNTERS];
long int cur_counter;

#define CounterInit()  cur_counter = -1

#define NewCounter()  assert(cur_counter < NR_COUNTERS - 1); \
                      cur_counter++; \
                      counter_stack[cur_counter].count = 1; \
                      counter_stack[cur_counter].ok = 0;

#define CloseCounter()  cur_counter--;

#define CounterOK()     counter_stack[cur_counter].ok = 1
```

```
#define IsCounterOK()   counter_stack[cur_counter].ok

#define TheCounter()    (counter_stack[cur_counter].count)
#define IncCounter()    TheCounter()++
#define DecCounter()    TheCounter()--
#define SetCounter(i)   TheCounter() = i

/* Term stack */

ATerm term_stack[TSIZE];
long int tsp;

/* The stackpointer tsp points to the element on top of the stack */

#define Ttop()       (term_stack[tsp])
#define Ttopi(i)     (term_stack[tsp - (i)])
#define Tset(t)      assert((t) != NULL); Ttop() = (t)
#define Tseti(i, t)  assert((t) != NULL); Ttopi(i) = (t)
#define Tpush()      assert(tsp < TSIZE); tsp++; Ttop() = NULL
#define Tpop()       assert(tsp > 0); tsp--
#define Tdupl()      Tpush(); Tset(Ttopi(1))
#define Tduplinv()   Tseti(1, Ttop());
#define Tdrop()      Tseti(1, Ttop()); Tpop()
#define Tinitst()    tsp = 0; Ttop() = NULL

#define Tswap(i, j)  {ATerm t; t = Ttopi(i); Tseti(i, Ttopi(j)); Tseti(j, t);}

#define Arg(n)       Tpush(); Tset(ATgetArgument(Ttopi(1), n))

#define Argi(i, n)   Tpush(); Tset(ATgetArgument(Ttopi(i), n))

void TprintStack(void);
void TprintStackTop(int j, char *s);

/* Return stack */

void *return_stack[RSIZE];
long int rsp;

void *fail_address;
ATbool failed;

/* rsp points to the next *free* element of the return stack */

#define Rpush(l) \
  if(rsp >= RSIZE) { \
    ATfprintf(stderr, "Fatal error: loop detected" \
      " (rstack overflow; rsp = %d)\n", rsp); \
    assert(rsp < RSIZE); \
  } return_stack[rsp++] = &&l;
```

```
#define Return() goto *(return_stack[--rsp])

/* Environment stack */

ATerm environment[ESIZE];
long int esp;

/* esp points to the next *free* element in the environment */

#define Einit()    esp = 0;

#define Ei(i)      (esp - (i))

#define Eget(i)    (between(0, Ei(i), ESIZE), environment[Ei(i)])
#define Eset(i, t) (between(0, Ei(i), ESIZE), environment[Ei(i)] = (t))

#define Egeta(i)    (between(0, i, ESIZE), environment[i])
#define Eseta(i, t) (between(0, i, ESIZE), environment[i] = (t))

#define Enext()    {assert(esp < ESIZE); environment[esp++] = NULL;}
#define Eprev()    {assert(esp < ESIZE); environment[--esp] = NULL;}

#define Epush(i)   {int j; for(j = 0; j < i; j++) {Enext();}}
#define Epop(i)    {int j; for(j = 0; j < i; j++) {Eprev();}}

void EprintStack(void);

/* Display stack */

long int display[DSIZE];
int dsp;

long int Ed(int i, int o);

#define Egetd(i, o)    Egeta(Ed(i, o))
#define Esetd(i, o, t) Eseta(Ed(i, o), t)

#define D(i)  display

#define DgetEsp()      display_stack[dsp].esp
#define DsetEsp(i)     display_stack[dsp].esp = i

void Epushd(int i, int o);
void Epopd(int i, int o);
void Dinit(void);
void Drestore(void);

/* Matching */

void *_MatchVar(int i);
void *_MatchVard(int i, long int o);
void *_MatchInt(int i);
void *_MatchIntFC(int i, void *);
void *_MatchReal(double i);
void *_MatchRealFC(double i, void *);
void *_MatchString(char *s);
void *_MatchStringFC(char *s, void *);
void *_MatchFun(char *f, int n);
void *_MatchFunFC(char *f, int n, void *fc);

#define MatchVar(i)          GO(_MatchVar(i))
#define MatchVard(i, o)      GO(_MatchVard(i, o))
#define MatchInt(i)          GO(_MatchInt(i))
#define MatchIntFC(i, fc)    GO(_MatchIntFC(i, fc))
#define MatchReal(i)         GO(_MatchReal(i))
#define MatchRealFC(i, fc)   GO(_MatchRealFC(i, fc))
#define MatchString(s)       GO(_MatchString(s))
#define MatchStringFC(s, fc) GO(_MatchStringFC(s, fc))
#define MatchFun(f, n)       GO(_MatchFun(f, n))
#define MatchFunFC(f, n, fc) GO(_MatchFunFC(f, n, fc))

/* Building */

#define BuildVar(i) \
  {ATerm t; \
   if((t = Eget(i)) == NULL) { \
     /* ATfprintf(stderr, "Warning: unbound variable (%d)\n", i); */ \
     goto *fail_address; \
   } else {Tset(t);}}

#define BuildVard(i, o) \
  {ATerm t; \
   if((t = Egetd(i, o)) == NULL) { \
     /* ATfprintf(stderr, "Warning: unbound variable (%d, %d)\n", i, o); */ \
     goto *fail_address; \
   } else {Tset(t);}}

#define BuildStr(s) \
  Tset(ATmakeString(s))

#define BuildInt(i) \
  Tset((ATerm) ATmakeInt(i))

#define BuildReal(r) \
  Tset((ATerm) ATmakeReal(r))

void BuildFun(char *f, int i);

/* Annotations */
```

```
#define GetAnn() \
{ATerm x; \
 if((x = ATgetAnnotation(Ttopi(1), Ttop())) == NULL) \
  { goto *fail_address; } \
 else { Tset(x); } \
}

#define SetAnn() \
 Tseti(2, ATsetAnnotation(Ttopi(2), Ttopi(1), Ttop())); Tpop(); Tpop()

#define RemoveAnn() \
  Tseti(1, ATremoveAnnotation(Ttopi(1), Ttop())); Tpop();

/* Choice stack */

struct {
  long int tsp;
  long int esp;
  long int rsp;
  long int counter;
  void    *continuation;
  int      dsp;
} choice_stack[CSIZE];
long int csp;

/* csp points to the next free choice stack entry */

#define Cset(cont, a, b, c, d, e) \
{ \
  choice_stack[csp].continuation = cont; \
  choice_stack[csp].tsp = a; \
  choice_stack[csp].esp = b; \
  choice_stack[csp].rsp = c; \
  choice_stack[csp].counter = d; \
  choice_stack[csp].dsp = e; \
}

#define Cpush(x) \
{ \
  assert(csp < CSIZE); \
  Cset(&&x, tsp, esp, rsp, cur_counter, dsp); \
  csp++; Tdupl(); \
}

#define Cpop()  Tdrop(); csp--;

#define Crestore()  \
{ \
  csp--; \
  tsp = choice_stack[csp].tsp; \
  esp = choice_stack[csp].esp; \
  rsp = choice_stack[csp].rsp;\
  dsp = choice_stack[csp].dsp; \
  cur_counter = choice_stack[csp].counter; \
}

#define Cjump() {goto *choice_stack[csp].continuation;}

#define Cempty() (csp == 0)

/* Path Traversal */

#define IsAppl() if(!t_is_appl(Ttop())) {goto *fail_address;};

#define TpushIthSon(i) Tpush(); Tset(ATgetArgument(Ttopi(1), i - 1));

#define TputIthSon(i) Tseti(1, ATsetArgument(Ttopi(1), Ttop(), i - 1)); Tpop();

/* Traversal */

void TravInit();
void *_NextSon(void *on_empty);
void *_SomeNextSon(void *s);
void TravBuild(void);

#define NextSon(x)     GO(_NextSon(x))
#define SomeNextSon(x) GO(_SomeNextSon(x))

#define AllInit()      TravInit()
#define AllNextSon(s)  NextSon(s)
#define AllBuild()     TravBuild()

#define OneInit()      TravInit()
#define OneNextSon()   NextSon(fail_address)
#define OneBuild()     TravBuild()

#define SomeInit()     TravInit()
#define SomeBuild()    TravBuild()

#define ThreadInit()  \
   MatchFun("Pair", 2); \
   Arg(1); \
   Argi(2,0); \
   TravInit();

#define ThreadNextSon(s) \
   NextSon(s); \
   Tpush(); \
   Tset(Ttopi(TheCounter() + 3)); \
   Tpush(); \
   BuildFun("Pair", 2);

#define ThreadSetEnv() \
   MatchFun("Pair",2); \
   Arg(0); \
   Argi(2, 1); \
   Tseti(TheCounter() + 4, Ttop()); \
   Tpop(); \
   Tdrop();

#define ThreadGetEnv() ThreadSetEnv()

#define ThreadBuild()  \
   TravBuild(); \
   Tswap(0,1); \
   Tpush(); \
   BuildFun("Pair", 2); \
   Tdrop();

void MatchTravInit();
void *_MatchNextSon(void *on_empty);

#define OneMatchNextSon() GO(_MatchNextSon(fail_address))
#define MatchTravEnd()  {Tpop(); CloseCounter();}

/* Procedure header and footer */

#define DOIT_START \
\
ATerm doit(ATerm t) \
{ \
  Tinitst(); Einit(); CounterInit(); Tset(t); fail_address = &&fail;

#define DOIT_END \
\
  return(Ttop()); \
  fail : \
    if(!Cempty()) {Crestore(); Cjump();} \
    else {failed = ATtrue; return Ttop();} \
  exit(1); \
}

#endif
```

Chapter: Run-time System
Section: stratego.c

```
/*

Copyright (C) 1998, 1999 Eelco Visser <visser@acm.org>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2, or (at your option)
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
02111-1307, USA.

*/

/*

$Id: stratego.c,v 1.5 2000/06/12 14:04:53 visser Exp $

Implementation of abstract machine instructions for stragegy
primitives. See strategy.h for more documentation.

*/

#include <aterm2.h>
#include "stratego.h"

#define ATmakeSymbol ATmakeAFun

ATbool ATfindSymbol(char *name, int arity, ATbool quoted);

/* Profiling */

int rule_counter = 0;
int match_counter = 0;
int build_counter = 0;
ATbool failed = false;

/* Term stack */

void TprintStack(void)
{
  TprintStackTop(tsp, "");
}
```

```
void TprintStackTop(int j, char *s)
{
  int i;
  ATfprintf(stderr, "tsp = %d %s\n", tsp, s);
  for(i = tsp; i >= tsp - j && i >= 0; i--) {
    if(term_stack[i] == NULL)
      ATfprintf(stderr, "  ts[%d] = NULL (%d)\n", i, tsp == i);
    else
      ATfprintf(stderr, "  ts[%d] = %t (%d)\n",   i, term_stack[i], tsp == i);
  }
}

/* Environment stack */

void EprintStack(void)
{
  int i;
  ATfprintf(stdout, "esp = %d\n", esp);
  for(i = 0; i <= esp; i++) {
    if(environment[i] == NULL)
      ATfprintf(stdout, "  es[%d] = NULL (%d)\n", i, esp == i);
    else
      ATfprintf(stdout, "  es[%d] = %t (%d)\n",   i, environment[i], esp == i);
  }
}

/* Display stack */

int max_nesting = -1;

#define Desp(x)     display[x]
#define Dnesting(x) display[x - 1]
#define Dnext(x)    display[x - 2]

void Dinit(void)
{
  dsp = -1;
}

int dframe(int i)
{
  int x;
  x = dsp;
  while(x >= 2 && Dnesting(x) > i)
    x = Dnext(x);
  return Dnesting(x) == i ? x : -1;
}

long int Ed(int i, int o)
```

```
{
  int x;
  x = display[dframe(i)] - o;
  //ATfprintf(stderr, "Ed(%d, %d) = %d (frame = %d)\n", i, o, x, dframe(i));
  return x;
}

void Epushd(int i, int o)
{
  int x = dframe(i - 1);
  Epush(o);
  dsp += 3;
  Desp(dsp)     = esp;
  Dnesting(dsp) = i;
  Dnext(dsp)    = x;
  //ATfprintf(stderr, "Epushd(%d, %d) : x = %d dsp = %d\n", i, o, x, dsp);
}

void Epopd(int i, int o)
{
  assert(Desp(dsp) == esp);
  assert(Dnesting(dsp) == i);
  Epop(o);
  dsp -= 3;
  //ATfprintf(stderr, "Epopd(%d, %d) : dsp = %d\n", i, o, dsp);
}

void Drestore(void)
{
}

/* Matching */

void *_MatchVar(int i)
{
  if(Eget(i) == NULL) {Eset(i, Ttop());}
  else if(!ATisEqual(Eget(i), Ttop())) {return fail_address;}
  return NULL;
}

void *_MatchVard(int i, long int o)
{
  if(Egetd(i, o) == NULL) {Esetd(i, o, Ttop());}
  else if(!ATisEqual(Egetd(i, o), Ttop())) {return fail_address;}
  return NULL;
}

void *_MatchInt(int i)
{
  if((ATgetType(Ttop()) != AT_INT) || (ATgetInt((ATermInt) Ttop()) != i))
```

```
    {return fail_address;}
  return NULL;
}

void *_MatchIntFC(int i, void *fc)
{
  if((ATgetType(Ttop()) != AT_INT) || (ATgetInt((ATermInt) Ttop()) != i))
    {return fc;}
  return NULL;
}

void *_MatchReal(double i)
{
  if((ATgetType(Ttop()) != AT_REAL) || (ATgetReal((ATermReal) Ttop()) != i))
    {return fail_address;}
  return NULL;
}

void *_MatchRealFC(double i, void *fc)
{
  if((ATgetType(Ttop()) != AT_REAL) || (ATgetReal((ATermReal) Ttop()) != i))
    {return fc;}
  return NULL;
}

void *_MatchString(char *s)
{
  if(!ATisThisString(Ttop(), s))
    {return fail_address;}
  return NULL;
}

void *_MatchStringFC(char *s, void *fc)
{
  if(!ATisThisString(Ttop(), s))
    {return fc;}
  return NULL;
}

void *_MatchFun(char *f, int n)
{
  assert(Ttop() != NULL);
  //ATfprintf(stderr, "_MatchFun(%s, %d)\n", f, n);
  if(!t_is_appl(Ttop())
      || ATgetArity(ATgetSymbol(Ttop())) != n
      || strcmp(ATgetName(ATgetSymbol(Ttop())), f) != 0
      || ATisQuoted(ATgetSymbol(Ttop())))
    {return fail_address;}
  return NULL;
}
```

```
void *_MatchFunFC(char *f, int n, void *fc)
{
  assert(Ttop() != NULL);
  //ATfprintf(stderr, "_MatchFunFC(%s, %d)\n", f, n);
  if(!t_is_appl(Ttop())
     || ATgetArity(ATgetSymbol(Ttop())) != n
     || strcmp(ATgetName(ATgetSymbol(Ttop())), f) != 0
     || ATisQuoted(ATgetSymbol(Ttop())))
    {
      return fc;
    }
  return NULL;
}

/* Building */

void BuildFun(char *f, int i)
{
  int j;
  Tset((ATerm) ATmakeList0());
  for(j = i; j > 0; j--) {
    Tseti(1, (ATerm) ATinsert((ATermList) Ttop(), Ttopi(1)));
    Tpop();
  }
  Tset((ATerm) ATmakeApplList(ATmakeSymbol(f, i, ATfalse),
      (ATermList) Ttop()));
}

/* Traversal

Initialization: Declare a new counter that will keep track of the
number of arguments of the node. Push the list of arguments on the
stack and set the counter to 0. */

void TravInit()
{
  NewCounter();
  if(t_is_appl(Ttop())) {
    Tpush();
    Tset((ATerm) ATgetArguments((ATermAppl) Ttopi(1)));
  }
  else {Tpush(); Tset((ATerm) ATmakeList0());}
  SetCounter(0);
}

/* When the list of arguments is empty proceed with the code at
|on_empty|. Otherwise shift the next argument on the stack.  */

void *_NextSon(void *on_empty)
```

```
{
  if(ATisEmpty((ATermList) Ttopi(TheCounter())))
    {return on_empty;}
  else {
    Tpush();
    IncCounter();
    Tset(ATgetFirst((ATermList) Ttopi(TheCounter())));
    Tseti(TheCounter(), (ATerm) ATgetNext((ATermList) Ttopi(TheCounter())));
  }
  return NULL;
}

void *_SomeNextSon(void *s)
{
  if(ATisEmpty((ATermList) Ttopi(TheCounter())))
    {if(IsCounterOK()) {return s;} else {return fail_address;}}
  else {
    Tpush();
    IncCounter();
    Tset(ATgetFirst((ATermList) Ttopi(TheCounter())));
    Tseti(TheCounter(), (ATerm) ATgetNext((ATermList) Ttopi(TheCounter())));
  }
  return NULL;
}

/* Rebuild the list of arguments and rebuild the application of the
original function symbol. */

void TravBuild(void)
{
  if(t_is_appl(Ttopi(TheCounter() + 1)))
    {
      /* if(!(ATgetArity(ATgetSymbol(Ttopi(TheCounter() + 1)))
              == TheCounter() + ATgetLength(Ttopi(TheCounter()))))
{
  int i;
  for(i = 0; i <= TheCounter() + 1; i++)
    {
      ATfprintf(stdout, "tsp[top-%d] = %t\n", i, Ttopi(i));
    }
  ATfprintf(stdout, "TheCounter = %d arity = %d\n", TheCounter(),
    ATgetArity(ATgetSymbol(Ttopi(TheCounter() + 1))));
}
      */
      assert(ATgetArity(ATgetSymbol(Ttopi(TheCounter() + 1))) ==
             (TheCounter() + ATgetLength(Ttopi(TheCounter()))));
      for(; TheCounter() > 0; DecCounter())
{
  Tseti(TheCounter(),
(ATerm) ATinsert((ATermList) Ttopi(TheCounter()), Ttop()));
```

```
  Tpop();
}
      Tseti(1,
    (ATerm) ATmakeApplList(ATgetSymbol(Ttopi(1)), (ATermList) Ttop()));
    }
  Tpop();
  CloseCounter();
}

/* Matching Traversal

When a traversal looks into a term only to match subterms, there is no
need to rebuild the term afterwards. */

void MatchTravInit()
{
  NewCounter();
  SetCounter(ATgetArity(ATgetSymbol(Ttop())));
  Tpush();
}

/* When the list of arguments is empty proceed with the code at
|on_empty|. Otherwise shift the next argument on the stack.  */

void *_MatchNextSon(void *on_empty)
{
  if(TheCounter() == 0)
    {return on_empty;}
  else {
    DecCounter();
    Tset(ATgetArgument(Ttopi(1), TheCounter()));
  }
  return NULL;
}
```

Chapter: Other Operations
Section: Experiments

BIBLIOGRAPHY

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

[2] 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.

[3] 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.

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

[5] 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.

[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 P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.

[10] Eelco Visser and Zine-el-Abidine Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15, September 1998. In C. Kirchner and H. Kirchner, editors, Proceedings of the Second International Workshop on Rewriting Logic and its Applications (WRLA'98), Pont-à-Mousson, France.

[11] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).