Appalachian STATE UNIVERSITY
BOONE, NORTH CAROLINA

Warm Fusion In Stratego: A Case Study in Generation Of Program Transformation Systems

By: **Patricia Johann** and Eelco Visser

Abstract found on next page

Johann, Patricia & Visser, Eelco **(2000)** "Warm Fusion In Stratego: A Case Study In Generation of Program Transformation Systems" *Annals of Mathematics and Artificial Intelligence*, vol. 29, no. 1-4 (2000), pp. 1 - 34. (ISSN: 1573-7470) Version Of Record Available From www.springer.com

Summary

Stratego is a domain-specific language for the specification of program transformation systems. The design of Stratego is based on the paradigm of rewriting strategies: user-definable programs in a little language of strategy operators determine where and in what order transformation rules are (automatically) applied to a program. The separation of rules and strategies supports modularity of specifications. Stratego also provides generic features for specification of program traversals.

In this paper we present a case study of Stratego as applied to a non-trivial problem in program transformation. We demonstrate the use of Stratego in eliminating intermediate data structures from (also known as *deforesting*) functional programs via the *warm fusion* algorithm of Launchbury and Sheard. This algorithm has been specified in Stratego and embedded in a fully automatic transformation system for kernel Haskell. The entire system consists of about 2600 lines of specification code, which breaks down into 1850 lines for a general framework for Haskell transformation and 750 lines devoted to a highly modular, easily extensible specification of the warm fusion transformer itself. Its successful design and construction provides further evidence that programs generated from Stratego specifications are suitable for integration into real systems, and that rewriting strategies are a good paradigm for the implementation of such systems.

This report contains the complete Stratego specification of the transformation. The first chapter, which will appear as a selfcontained publication, explains the ideas of the transformation, gives an overview of the specification and discusses several techniques used in the specification. The subsequent chapters present the specification of syntax of the language, basic operations, typechecking, simplification and the actual transformation. In addition to the abstract syntax, a concrete syntax definition in SDF2 is given as an example of connection of a parser frontend to transformation systems built with Stratego.

Contents

Chapter 1

Warm Fusion in Stratego

Stratego is a domain-specific language for the specification of program transformation systems. The design of Stratego is based on the paradigm of rewriting strategies: user-definable programs in a little language of strategy operators determine where and in what order transformation rules are (automatically) applied to a program. The separation of rules and strategies supports modularity of specifications. Stratego also provides generic features for specification of program traversals.

In this paper we present a case study of Stratego as applied to a non-trivial problem in program transformation. We demonstrate the use of Stratego in eliminating intermediate data structures from (also known as *deforesting*) functional programs via the *warm fusion* algorithm of Launchbury and Sheard. This algorithm has been specified in Stratego and embedded in a fully automatic transformation system for kernel Haskell. The entire system consists of about 2600 lines of specification code, which breaks down into 1850 lines for a general framework for Haskell transformation and 750 lines devoted to a highly modular, easily extensible specification of the warm fusion transformer itself. Its successful design and construction provides further evidence that programs generated from Stratego specifications are suitable for integration into real systems, and that rewriting strategies are a good paradigm for the implementation of such systems.

1.1 Introduction

Automatic program transformation is applied in many branches of software engineering — including application generation and compiler construction — to translate high-level, but inefficient, specification code to lower-level and more efficient implementation code. It plays a particularly important role in compilers for functional programming languages [9, 3, 10, 25, 28].

1.1.1 Transforming Programs with Rewriting Strategies

An important paradigm for the description of program transformation systems is that of rewrite rules. Ad-hoc implementation of transformation systems based on rewrite rules can be difficult, however, because the rules must be embedded in

algorithms that determine strategies for applying them. Stratego [17, 35, 36, 32] is a domain-specific language for the specification of program transformation systems. Its design is based on the paradigm of rewriting strategies. Rewriting strategies combine user-definable rewriting-based programs with a little language of independent strategy operators that can be used to specify where and in what order transformation rules are applied to a program.

Stratego's separation of rewrite rules from the strategies which control their application facilitates modular specification of program transformations: transformation rules are specified independently of the application strategy and can be reused in more than one strategy. Stratego also offers both fine and coarse grain control over the application of transformation rules. This control makes it possible to specify the exact forms that programs can assume at various stages of processing. It also allows the programmer to govern the interactions between individual transformation rules. The Stratego compiler translates specifications to C programs that transform abstract syntax trees to abstract syntax trees.

In [36] it is shown how rewriting strategies can be used to modularly specify and implement optimizers for functional programs. A set of transformation rules is combined into a code simplification algorithm by means of a strategy that traverses programs and applies rules where appropriate. The emphasis in [36] is on rules that are independently applicable. As demonstrated there, it is particularly easy to combine transformation rules into different simplification strategies by adding or omitting rules. But in many settings the construction of interrelated transformation rules from several more primitive rules is necessary.

1.1.2 Applying Strategies in Deforesting Functional Programs

In this paper we present a case study illustrating the use of rewriting strategies to eliminate intermediate data structures from (deforest) functional programs. Deforestation algorithms typically perform a number of smaller transformations before determining whether or not the deforestation is considered successful. Combining primitive rules into complex program transformations often requires the exchange of more information between their rules than is contained in the individual program fragments they transform. The parameterization of strategies supported by Stratego provides a means of specifying and implementing rules which pass such information between them. In the case study presented here information exchanged between transformations takes the form, for example, of assumptions about bindings, dynamic rewrite rules that recognize recursive function calls, and terms generated by splitting functions to facilitate program transformation. Parameterized strategies have not been used extensively in previous Stratego specifications.

We have specified the warm fusion algorithm of Launchbury and Sheard [16] in Stratego. This technique combines the cheap deforestation based on foldr-build fusion of Gill et al. [11, 10] with the fold promotion of Sheard and Fegaras [26] and a generalization of the technique of Peyton Jones and Launchbury [24] for splitting a function into a worker and a wrapper. The foldr-build fusion, which has been implemented in the Glasgow Haskell Compiler (GHC), requires the manual transformation of functions to build-foldr form and is only defined for lists. The warm fusion algorithm generalizes cheap deforestation to arbitrary regular data types and automatically derives more general build-cata forms.

As a case study, the warm fusion algorithm is an interesting example of a non-trivial program transformation, and its specification provides evidence of the feasibility of implementing such transformations in Stratego. The case study supplies experience with the design and implementation of a complete transformation system, including interfaces with a parsing and typechecking front-end and a pretty-printing back-end for Haskell. The application to Haskell provides an environment in which to assess the effectiveness of warm fusion for deforesting more realistic programs than would otherwise be possible. The case study also demonstrates Stratego's support for the construction of transformation rules that combine basic transformation steps in various ways, the description and checking of intermediate representation formats, language independent definition of substitution and the renaming of bound variables, and the discovery of new programming idioms resulting from the strategy-induced shift away from a purely functional implementation style.

Warm fusion is also an interesting problem in its own right. The first fully automatic implementation of warm fusion was hand-coded in Haskell in 1997 [14]. The algorithm had previously been implemented only as 'a toolbox of operations' [16]. This is perhaps because the description of warm fusion in [16] elides much of the detail required to turn the theory into practice. The type-driven nature of the algorithm, in particular, is fundamental to its automation, as well as to its extension to non-list data structures. The critical dependence of warm fusion on type information is reflected in its Stratego specification.

The product of our case study is a fully automatic implementation of the warm fusion algorithm. This implementation could be an important step toward the use of warm fusion in compilers or as a preprocessor for (library) programs. It can also serve as a basis for further experimentation with extensions of cheap deforestation; Stratego makes it easy, for example, to modify the set of program transformation rules and to experiment with a variety of application orders. Experience with a working system often gives rise to a deeper understanding of its underlying algorithm. It was such experience that led, for instance to our "double splitting" wrapper-worker technique for recognizing certain variables as static parameters of programs undergoing warm fusion. (This step happens "automagically" in [16]). This technique has since been incorporated into the Haskell implementation of warm fusion detailed in [14].

1.1.3 Outline

In the next section we briefly review some background on deforestation, discuss the principles of cata-build fusion, and illustrate the warm fusion transformation technique by means of an example. In Section 1.3 we give an overview of the operators of System S, a calculus for the definition of tree transformations, as well as of the syntactic abstractions built on System S that form Stratego. In Section 1.4 we present the overall architecture of the warm fusion transformation tool built with Stratego. In Sections 1.5, 1.6, and 1.7 we discuss several highlights from the specification, focusing particularly on some of the new programming idioms that have emerged during the process of specifying the warm fusion algorithm in Stratego. The full text of the specification can be found in the next chapters.

```
data Bool = True | False;
data List a = Nil | Cons a (List a);
map :: (a -> b) -> List a -> List b;
map = \f l ->
      case l of {
        Nil       -> Nil;
        Cons x xs -> Cons(f x)(map f xs)};
foldr :: b -> (a -> b -> b) -> List a -> b;
foldr = \n c xs ->
        case xs of {
          Nil       -> n;
          Cons y ys -> c y (foldr n c ys)};
upto :: Int -> Int -> List Int;
upto = \low high ->
       case low > high of {
         True  -> Nil;
         False -> Cons low (upto(low + 1)(high))};
sum :: List Int -> Int;
sum = foldr 0 (+);
sos :: Int -> Int -> Int;
sos = \lo hi -> sum(map(square)(upto lo hi))
```

Figure 1.1: Recursive functions on lists

1.2 Warm Fusion

Modularity in functional programming is achieved by dividing programs into small, generally applicable functions that communicate via data structures. Such functions are commonly defined as recursive operations that construct and deconstruct data structures. The definitions in Figure 1.1 are common examples of such functions; `sum` and `foldr` consume lists, `upto` produces lists, and `map` does both. Using these functions we can, for instance, define the sum of the squares of the numbers `lo` to `hi` as

```
sos :: Int -> Int -> Int
sos = \lo hi -> sum(map(square)(upto lo hi))
```

where the function `square` is defined as

```
square :: Int -> Int
square = \x -> (x * x)
```

This implementation of the sum-of-squares function is straightforward and modular. Its disadvantage is that it constructs, traverses, and deconstructs two intermediate lists — even though both the input and output of the computation are integers. This is computationally expensive, both slowing execution time and increasing heap space requirements.

It is often possible to avoid manipulating intermediate data structures by using a more elaborate style of programming in which parts from component functions are intermingled. In this monolithic style of programming the sum-of-squares function is defined as

```
sos' :: Int -> Int -> Int
```

```
sos' = \lo hi ->
       let {sos'' :: Int -> Int;
            sos'' = \i -> case i > hi of {
                            True  -> 0;
                            False -> square(i) + sos''(i + 1)}}
       in sos''(lo)
```

Note that no intermediate data structures at all are processed by `sos'`. In this case, eliminating the manipulation of intermediate lists results in an order of magnitude gain in program performance.

Experienced programmers writing a square summing function would instinctively produce `sos'` rather than `sos`; small functions like `sos` are easily optimized at the keyboard. But when programs are either very large or very complex, even experienced programmers may find that eliminating intermediate data structures by hand is not a very attractive alternative to the modular style of programming. In such situations a tool for automatically eliminating them is needed.

1.2.1 Deforestation

Automatic elimination of intermediate data structures by transformation combines the clarity and maintainability of the modular style of programming with the efficiency of the monolithic style. The process of eliminating intermediate data structures from programs is often called *deforestation* after an early transformation technique of Wadler [37] which removes tree-like data structures from first-order programs.

In Wadler's deforestation, compositions of treeless expressions (a syntactic restriction of normal expressions that allows no intermediate data structures) are transformed into new treeless expressions. The technique uses function unfolding to expose consumption of constructors by case selections. Subsequent folding creates new recursive functions. To prevent non-termination of unfolding, global program patterns must be monitored. Because this is computationally expensive, Wadler's deforestation has not been incorporated into functional language compilers.

Gill et al. [10, 11] introduce a less general, but cheaper, variant of deforestation for list-producing and -consuming functions. The key observation underlying their *short cut* to deforestation is that many list-manipulating functions can be written in terms of the uniform list-consuming function `foldr` and the uniform list-producing function `build`. Since `foldr` is another name for the standard catamorphism for lists, we denote it by `cata-list` in this paper. And since the `build` function of Gill et al. is the instantiation to lists of a more general `build` function applying to arbitrary regular data types, we denote it by `build-list` below.

Operationally, `cata-list` takes as input types `a` and `b`, a replacement function `f1 :: a -> b -> b` for `Cons[a]`, a replacement function `f2 :: b` for `Nil[a]`, and a list `ls` of type `List a`. (The list constructors `Cons` and `Nil` have the polymorphic types `forall a. a -> List a -> List a` and `forall a. List a`, respectively, and so must be instantiated for each particular list type; the notation `e[t]` instantiates the polymorphic expression `e` to type `t`.) It replaces by `f1` and `f2`, respectively, all occurrences of `Cons[a]` and `Nil[a]` in `ls` which

```
map :: (a -> b) -> List a -> List b;
map = \f l ->
      build[List b](/\t -> \(n :: t) (c :: (b -> t -> t)) ->
                     cata[List a][t](n, \(y :: b) -> c(f y)) l);

foldr :: b -> (a -> b -> b) -> List a -> b;
foldr = \n c -> cata[List a][b](n, c);

upto :: Int -> Int -> List Int;
upto = \lo  hi ->
       build[List Int]
            (/\t -> \(n :: t) (c :: int -> t -> t) ->
             let {upto':: Int -> t;
                  upto' = \i -> case i > hi of {
                                  True -> n;
                                  False -> c(i)(upto'(i + 1))}}
              in upto'(lo));

sum :: List Int -> Int;
sum = cata[List Int][Int](0, (+))
```

Figure 1.2: Functions in build-cata form

actually contribute to the result of the computation. The result is a value of type `b`. The function `build-list`, on the other hand, takes as input a function `g` providing a type-independent template for constructing lists and instantiates its "abstract" list constructors with appropriate instances of the "concrete" list constructors `Cons` and `Nil`. In other words, if `g` is any function with polymorphic type `forall b . b -> (a -> b -> b) -> b`, then

```
build-list[a](g) = g[List a] (Nil[a]) (Cons[a])
```

Compositions of list-consuming and -producing functions defined in terms of `cata-list` and `build-list` can be simplified (deforested) by means of the short cut fusion rule for lists:

```
cata-list[a][t](f1, f2)(build-list[a](g)) = g[t] f1 f2
```

The short cut describes one precise way in which compilers can take advantage of uniformity in the production and consumption of lists to optimize programs which manipulate them. It makes sense intuitively: the result of a computation is the same regardless of whether the function `g` is first applied to `Cons` and `Nil` and occurrences of `Cons` and `Nil` in the resulting list are then replaced by `f1` and `f2`, respectively, or the abstract constructors in `g` are replaced by `f1` and `f2`, respectively, directly. The fact that `g` is polymorphic in its result type `t` ensures the correctness of this fusion rule.

1.2.2 An Example of Cata-Build Fusion

Figure 1.2 shows the build-cata forms of the functions in Figure 1.1. The notation `/\a -> e` denotes the abstraction of type variable `a` from the expression `e`. Such an expression has type `forall a . t`, where `t` is the type of `e`. Type abstraction is normally implicit in definitions in Haskell because it only occurs at

the top of a definition, i.e., a Haskell definition `f = \x -> e` that is polymorphic in type variable `a` abbreviates the definition `f = /\a -> \x -> e`.

The deforested function `sos'` can be derived from `sos` by inlining the definitions in Figure 1.2 and applying the short cut in conjunction with the standard program simplification rules in Section 1.7. Inlining the (type-instantiated) function definitions for `sum`, `map` and `square` gives

```
sos = \lo hi -> sum(map(square)(upto lo hi))
    = \lo hi ->
      cata[List Int][Int](0, (+))
      ((\f l -> build[List Int]
                 (/\t -> \(n :: t)  (c :: Int -> t -> t) ->
                  cata[List Int][t](n, \(y :: Int) -> c(f y)) l))
       (\x -> x * x) (upto lo hi))
```

Simplifying the application of `map` to `square` and `upto lo hi` produces

```
    = \lo hi ->
      cata[List Int][Int](0, (+))
      (build[List Int]
       (/\t -> \(n :: t)  (c :: Int -> t -> t) ->
        cata[List Int][t](n, \(y :: Int) -> c(y*y))(upto lo hi))
```

Applying the short cut rule to the `cata-build` pair and simplifying yields

```
    = \lo hi ->
      cata[List Int][Int](0, \(y :: Int) -> (+)(y*y))(upto lo hi)
```

Inlining the definition for `upto` gives

```
    = \lo hi ->
      cata[List Int][Int](0, \(y :: Int) -> (+)(y*y))
      (build[List Int]
       (/\t -> \ (n :: t) (c :: Int -> t -> t) ->
        let {upto'::Int -> t;
             upto' = \i -> case i > hi of {
                             True -> n;
                             False -> c(i)(upto'(i+1))}}
         in upto'(lo)))
```

Using the short cut and simplifying once more gives

```
 sos = \lo hi ->
       let {upto'::Int -> Int;
            upto' = \i -> case i > hi of {
                            True -> 0;
                            False -> (i*i) + (upto'(i+1))}}
        in upto'(lo)
```

Up to renaming and inlining of square in the local function, this is precisely the definition of `sos'`.

1.2.3 Warm Fusion: Automatically Deriving Cata-Build Forms

The short cut fusion rule calculates program improvement based on a program's explicit local structure. To do this, it requires that functions be written in the highly stylized build-cata form, rather than using explicit recursion. But this is often not the most natural way to develop programs. Moreover, because `build` does not have a Hindley-Milner type — and so can only be used in certain well-defined ways — providing it for programmers' direct use is problematic. The warm fusion algorithm of Launchbury and Sheard [16] was designed to automate the safe introduction of `build` into recursive list-processing functions, as well as the transformation of the resulting functions into equivalent ones in build-cata form.

The existence of a catamorphism and a `build` function for each regular data type makes it possible to generalize the warm fusion method to arbitrary regular data types. If `F` is a functor defining a regular data type, then the catamorphism `cata[F a1...an][t](f1,...,fn)` replaces the constructors of a data structure of type `F a1...an` with the functions `fi`. The result of the catamorphism has type `t`. The data structure-producing function `build[F a1...an]`, on the other hand, takes as input a polymorphic function `g` which constructs the kind of data structures associated with the functor `F`. It replaces the abstract data constructors of `g` by the concrete data constructors `ci` to produce the data structure of type `F a` whose description `g` embodies. That is,

```
build[F a1...an](g) = g[F a1...an] c1 ... cn.
```

Note that `cata-list[a][t]` is just `cata[List a][t]` and `build-list[a]` is precisely `build[List a]`, where `List` is the functor associated with the list data type. The short cut fusion rule for `cata-list` and `build-list` generalizes to:

```
cata[F a1...an][t](f1,...,fn)(build[F a1...an](g)) = g[t] f1...fn
```

1.2.4 Warm Fusion by Example

To illustrate the process of warm fusion we will examine the transformation of the consumer-producer `map`. We refer the reader to [16] for theoretical justification of the method. In the following examples we will omit the type declarations for variables and constructors when these are clear from the context or from previous declarations.

Abstracting from Constructors The goal of the preprocessing step of warm fusion is to transform a recursive definition into a definition in build-cata form:

```
f = /\a1 ... an -> \x ...  ->
    build[F a1...an](/\t -> \c1...cn ->
                        cata[F a1...an][t](h1,...,hm) x)
```

The functional argument of `build` is a catamorphism that consumes the input data structure `x` and builds up a structure that is constructed with the abstract constructors `ci`. This transformation shifts the recursion boundary of the function from the site of construction of the result data structure to the site of

consumption of the input data structure. All recursion in build-cata forms is expressed via catamorphisms.

The first phase of the transformation abstracts away from the concrete constructors in the body of the function. This cannot be done simply by replacing all constructors in the body by variables, however, because not all occurrences of constructors necessarily contribute to the result of the computation. By applying `cata[F a1...an][t](c1,...,cn)` to the body of the function, the result-producing constructors are transformed into the corresponding abstract constructors `ci`.

The identity

```
x = build[F b1..bn](/\ t -> \c1 ... cn ->
                     cata[F b1..bn][t](c1, ..., cn) x)
```

is used to introduce this catamorphism to the body. For `map` this becomes

```
map = /\a b -> \f l ->
build[List b](/\t -> \(n :: t)  (c :: b -> t -> t) ->
cata[List b][t](n, c)(
  case l of {
    Nil -> Nil;
    Cons x xs -> Cons(f x)(map[a][b] f xs)))}
```

Distribution of the catamorphism over the case expression gives

```
map = /\a b -> \f l -> build[List b](/\t -> \n c ->
case l of {
  Nil -> cata[List b][t](n, c) Nil;
  Cons x xs -> cata[List b][t](n, c)(Cons(f x)(map[a][b] f xs)))}
```

Specialization of the catamorphism to the constructors that it is applied to produces:

```
map = /\a b -> \f l -> build[List b](/\t -> \n c ->
case l of {
  Nil -> n;
  Cons x xs -> c(f x)(cata[List b][t](n, c)(map[a][b] f xs)))}
```

Note that the catamorphism is applied to the recursive second argument of the abstract replacement function for `Cons`.

Splitting off the Recursive Consumer We have now abstracted away from the result-producing constructors of `map` and written it in the form of an abstracted call to `build`. Next we derive a catamorphism to replace the case analysis in `map`'s body. This is accomplished according to the steps outlined in the remainder of this section.

First the function body is split into two new definitions. For `map` we get the 'wrapper' `map` and the 'worker' `map#` (a generally applicable idea first presented in [24]):

```
map = /\a b -> \f l ->
      build[List b](/\t -> \n c -> map# l [t] n c)
map# = \l -> /\t -> \n c ->
```

```
case l of {
  Nil -> n;
  Cons x xs -> c(f x)(cata[List b][t](n, c)(map[a][b] f xs))}
```

The splitting has the effect of isolating a recursive definition not involving `build`.

Note that the function `f` and the type variables `a` and `b` are not passed to `map#`. From the definition of `map` before splitting it is clear that these arguments are passed unchanged to the recursive call of `map`. That is, they are *static parameters* of `map`. Since we do not abstract over them, the static parameters of a function remain free in the definition of its worker. This means that `f`, `a`, and `b` remain free in `map#`. When, at the end of the transformation, the transformed version of the function's worker is folded back into the definition of its wrapper, its static parameters will become bound again.

By unfolding the wrapper in the worker we obtain a recursive definition of the worker. For `map` we get

```
map# = \l -> /\t -> \n c ->
case l of {
  Nil -> n;
  Cons x xs ->
    c(f x)(cata[List b][t](n, c)
          ((/\a' b' -> \f' l' ->
            build[List b](/\t' -> \n' c' -> map# l' [t'] n' c'))
           [a][b] f xs))}
```

Beta-reduction and short cut fusion reduces this to

```
map# = \l -> /\t -> \n c ->
       case l of {
         Nil -> n;
         Cons x xs -> c(f x)(map# xs [t] n c)}
```

Observe now that all arguments except for `l` are static parameters of `map#`. By repeating the splitting and unfolding procedure once more we get

```
map#  = \l -> /\t -> \n c -> map## l
map## = \l -> case l of {
                 Nil -> n;
                 Cons x xs -> c(f x)(map## xs)}
```

The parameters `t`, `n`, and `c` of `map#` are now also recognized as static in `map##`. The free variable `f` in `map##` is inherited from `map`. In [16], mechanical recognition of the abstracted constructors as static parameters (when they are), happens magically.

Recursion to Catamorphism Finally, the recursive definition of `map##` is turned into a catamorphism by means of fold promotion. Fold promotion is based on a generic promotion theorem introduced by Malcolm [18]. The promotion theorem, which has its origins in a categorical description of programming [12], describes conditions under which the composition of an arbitrary (strict) function and a catamorphism over a regular data type may be fused to arrive at a new catamorphism equivalent to the original composition. For `map##` the promotion theorem takes the form

```
map## Nil = h1,
map##(Cons(y1, y2)) = h2(y1, map## y2)
----------------------------------------
map##(cata[List a][List a](Nil, Cons) xs)
  = cata[List a][t](h1, h2) xs
```

This means that we can find `h1` and `h2` by applying `map##` to `Nil` and `Cons y1 y2`, respectively, and abstracting from the recursive call to `map##`. For `Nil` this simply produces the abstracted constructor `n`. For `Cons` we get

```
h2 = \y1 y2 -> (\l -> case l of {
                  Nil -> n;
                  Cons x xs -> c(f x)(map## xs)})
                (Cons z1 z2)
```

where the `zi` are special constants. This reduces to

```
\y1 y2 -> c(f z1)(map## z2)
```

Now we use special rewrite rules generated from the type of the constructor to rewrite the dummy variables `zi` to the real variables `yi`. This makes it possible to discover the recursive invocation of the `map##` function and replace it by the induction variable. For the `Cons` constructor the rewrite rules `z1 -> y1` and `map## z2 -> y2` are generated. The first corresponds to an occurrence of the type parameter `a` and the second to a recursive occurrence of the type `List a`.

By application of the rewrite rules `z1 -> y1` and `map## z2 -> y2` the recursive call is recognized and we get

```
h2 = \y1 y2 -> c(f y1)(y2)
```

Putting this together gives the non-recursive definition

```
map## = \l -> cata[List a][t](n, \y1 y2 -> c(f y1)(y2)) l
```

Folding By unfolding the worker functions `map##` and `map#` back into their subsequent wrappers we obtain the build-cata form of `map`:

```
map = /\a b -> \f l -> build[List b](/\t -> \n c ->
      cata[List a][t](n, \y1 y2 -> c(f y1)(y2)) l)
```

Transforming Programs The transformation procedure illustrated above is attempted (it may fail) for all functions. Compositions of functions in build-cata form can be deforested by unfolding their definitions and applying short cut fusion as part of standard simplification (see Section 1.7). The unfolding can be done without risk of non-termination because the functions are not explicitly recursive.

The build-cata forms in Figure 1.2 are all obtained using this transformation. Note that not all of these functions do both produce and consume a list; `foldr` only consumes a list and `upto` only produces a list. Their cata-and-or-build forms are obtained using variants of the transformation process described above. These variants are discussed in Section 1.7 below.

We have specified the warm fusion transformation algorithm in Stratego. In the remainder of this paper we will give an overview of the specification. In particular, we will discuss the basic steps of the transformation such as splitting, unfolding, folding and deriving a catamorphism and how these can be used in various combinations and orders to obtain different results. First we give an overview of Stratego itself.

1.3 Stratego

In this section we briefly introduce System S, a calculus for the definition of tree transformations, and Stratego, a specification language providing syntactic abstractions for System S expressions. For a detailed description of Stratego, its operational semantics, and additional examples of its use we refer the reader to [1, 17, 35, 36, 32, 34]. Figure 1.3 shows a Stratego module defining several generic transformation operators. Other example specifications that use these operators will be discussed in the rest of the paper.

1.3.1 System S

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

Transformations in System S are applied to first-order terms, which are expressions over the grammar

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

where `x` ranges over variables and `C` over constructors. The notation `[t1,...,tn]` abbreviates the list `Cons(t1,...,Cons(tn,Nil))`. In addition, the notation `[t1,..,tn | t]` denotes `Cons(t1,...,Cons(tn,t))`.

Level 1: Sequential Non-deterministic Programming *Strategies* are programs that attempt to transform terms into 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 there is no result of the transformation. 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` of strategies `s1` and `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` of strategies `s1` and `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` of strategies `s1` and `s2` attempts to apply either `s1` or `s2`, in that order. The *recursive closure* `rec x(s)` of the strategy `s` attempts to apply `s` to the entire subject term and the strategy `rec x(s)` to each occurrence of the variable `x` in `s`. The *test* strategy `test(s)` tries to apply the strategy `s`. It succeeds if `s` succeeds, and reverts the subject term to the original term. It also 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

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

Figure 1.3: Specification of several generic term traversal strategies.

that can be defined with these operators are the `try` and `repeat` strategies in Figure 1.3.

Level 2: Term Traversal The combinators discussed above combine strategies that 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 a *congruence* operator `C(s1,...,sn)` for each `n`-ary constructor `C`. It applies to terms of the form `C(t1,...,tn)` and applies `si` to `ti`. An example of the use of congruences is the operator `map(s)` defined in Figure 1.3 that applies a strategy `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 1.3 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 we have 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 subject term matches with the term `t`. 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 behavior can also arise across 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.

Built-in Data types There are two predefined sorts with an infinite number of constructors: integers and strings. Several operators provide standard operations on these data types. Of particular importance for our purposes is the operator `new` that builds a new string that does not occur anywhere in the term being transformed.

1.3.2 Specifications

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. An example signature is shown in Figure 1.4. 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 `l` against the subject term, and then attempts to satisfy the condition `s`. If that succeeds, it builds the right-hand side `r`. The rule is enclosed in a scope that makes all term variables `xi` occurring freely 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`.

Strategy operators can only have strategies as arguments. Data can be passed to strategy operators by wrapping them in build expressions. For instance, the strategy `map(!A)` will replace every element of a list by the constant term `A`. Parameterized strategies have not often been used in previous Stratego specifications. They are nevertheless critical in specifying the warm fusion

```
module AHaskell
signature
  sorts Decl Constr Type Exp Alt
  operations
    Program    : List(Decl)                              -> Program

    Data       : Type * List(Constr) * Deriving          -> Decl
    ConstrDecl : Option(Forall) * Option(Context)
                 * String * List(Type)                   -> Constr

    SignDecl   : Vars * Type                             -> Decl
    Valdef     : Exp * Exp                               -> Decl

    TCon       : String                                  -> Type
    TVar       : String                                  -> Type
    TApp       : Type * List(Type)                       -> Type
    TFun       : List(Type) * Type                       -> Type
    Forall     : List(String) * Type                     -> Type

    Typed      : Exp * Type                              -> Exp
    Var        : String                                  -> Exp
    Constr     : String                                  -> Exp
    Lit        : Literal                                 -> Exp
    Abs        : List(Exp) * Option(Type) * Exp          -> Exp
    App        : Exp * List(Exp)                         -> Exp
    Let        : List(Decl) * Exp                        -> Exp
    Case       : Exp * List(Alt)                         -> Exp
    Alt        : Exp * Option(Type) * Exp                -> Alt
    TAbs       : List(String) * Exp                      -> Exp
    TInst      : Exp * List(Type)                        -> Exp

    Build      : Type * Exp                              -> Exp
    Cata       : Type * Type * List(Exp)                 -> Exp
```

Figure 1.4: Signature for kernel Haskell.

transformer and in other situations in which information must be passed between strategies.

The following definitions provide a useful shorthand. The notation `<s> t` denotes `!t; s`, i.e., the strategy which builds the term `t` and then applies `s` to it. The notation `s => t` denotes `s; ?t`, i.e., the strategy which 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 inside a term in a build expression. For example, the strategy expression `!F(<s> t, t')` corresponds to `{x: <s> t => x; !F(x,t')}`, where `x` is a new variable.

1.3.3 Derived Idioms

Stratego's syntactic abstractions give rise to a number of useful programming idioms. Foremost among these are *recursive patterns* and *distributed patterns*.

Recursive patterns are strategy expressions that describe term formats by means of congruences and recursion. Nested congruences in Stratego are similar to pattern matching in functional languages, and Stratego's recursive patterns involving nested patterns are akin to recursive functions which verify the structure of terms. Like pattern matching in functional languages, Stratego's recursive patterns are completely general. For example, the following recursive pattern describes the subset of Haskell expressions that corresponds to untyped λ-calculus terms:

```
lambda-exp =
  rec x(Var(id) + App(x, x) + Abs([Var(id)], None, x))
```

Their use is further demonstrated in the term format checking in Section 1.5. They can also be used to characterize more complicated formats such as normal forms or expressions in a core language. More generally, recursive patterns can be used whenever expressions in a sublanguage of a larger representation language must be recognized or manipulated.

Distributed patterns combine the pattern matching of recursive patterns with the traversal capabilities of strategy operators. They serve as "pattern templates" that can be used to match against expressions containing specified subexpressions at variable depths within them. For example, the warm fusion transformer uses the distributed pattern `underabstr` to determine whether or not a term in the expression language of Figure 1.4 contains an application whose argument term is an abstraction in which the variable (determined by the strategy) `s` appears:

```
underabstr(s) = oncetd(App(id, Abs(id,id,oncetd(Var(s)))))
```

Note that the argument term to the abstraction need not actually *be* the variable determined by `s`; all that is required is that the variable appear somewhere within the argument term. More general distributed patterns are constructed with the same ease.

1.3.4 Implementation

The Stratego compiler translates a specification to a C program that reads a term and applies the specified transformation to it. The compiler first translates

a specification to a System S expression, which is then translated to a list of abstract machine instructions. The instructions are implemented in C. The run-time system is based on the ATerm library [4], which supports complete sharing of subterms (hash-consing). ATerms are also used for exchange of data between components of a transformation systems. The compiler is bootstrapped, i.e., implemented in Stratego itself. The Stratego library [33] provides a large of number generic, language independent rules and strategies.

1.4 Architecture

The architecture of the warm fusion program transformation system is depicted in Figure 1.5. The system consists of four main components: a parser, typechecker, the actual warm fusion transformer, and a pretty-printer. The system could have been defined as a single component, but dividing it into separate components encourages separation of concerns during development and makes future application of the transformation tool in another setting — e.g., connection to a compiler front-end — easier.

The parser is generated from a specification of the full[1] Haskell98 syntax [23] in the syntax definition formalism SDF2 [31]. Although the parser supports the full syntax, currently only the kernel subset of Haskell is supported by the subsequent components. A Haskell desugaring component can be added in the future to extend the transformer to full Haskell.

Note that SDF2 based parsers are not required for Stratego. Parsing front-ends can also be written using YACC or any other parser generator, as long as the generated parsers output abstract syntax trees in the ATerm format. The SDF2 parser that we use actually outputs parse trees. These are transformed to abstract syntax trees by a generic — i.e., grammar independent — tool (implode-asfix) written in Stratego.

The current typechecker is basically a preprocessor that distributes type information from signature declarations to variable uses. This could be enhanced to a tool that does full type inference, but for the purposes of our case study this was not necessary; types of variables are declared explicitly in input programs. Note that this is not too much of a restriction. In Haskell it is customary to declare the types of functions anyway.

The intermediate data structures that are exchanged between components are represented in the generic ATerm format [4]. Furthermore, each component consumes and produces a different subset of the general abstract syntax of the language. These formats are also described in Stratego by means of strategies that check the structure of a term. These strategies can be used by components to verify their input.

The warm fusion transformer processes each of the function definitions in a program and tries to transform it into build-cata form. It also inlines previously transformed functions in the definitions it is processing to achieve deforestation by the short cut.

The pretty-printer is a formatter that translates abstract syntax to strings. A Stratego specification (PP-Haskell) defines the translation from abstract syntax to Box terms. These are translated to formatted text by a generic Box formatter [5, 15].

[1] The syntax definition is complete up to layout.

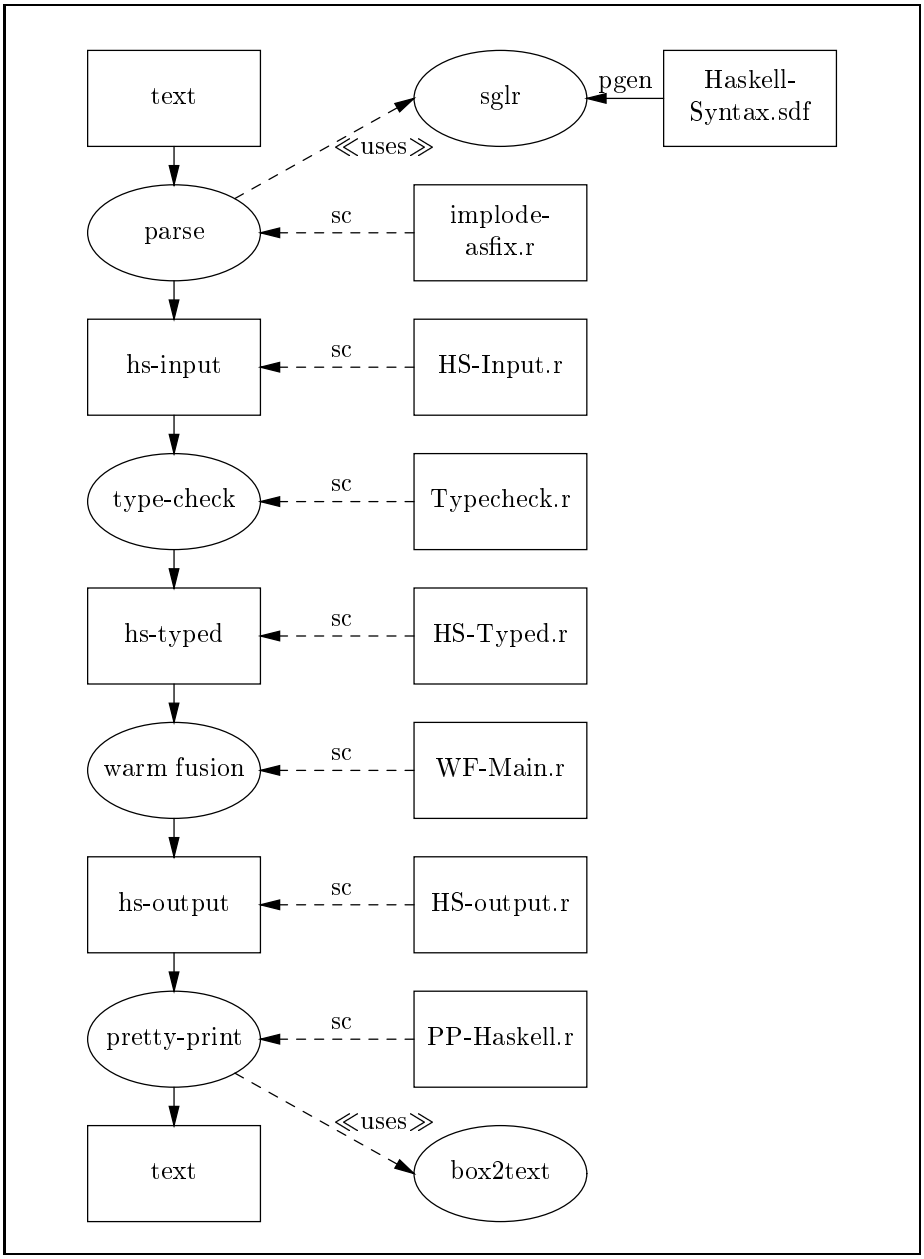

Figure 1.5: Architecture of the warm fusion transformation tool. Boxes represent data, ellipses represent components. Dashed arrows represent generation of components from specifications via the Stratego compiler (sc), the SDF2 parser generator (pgen) and a C compiler (gcc). The intermediate data-formats are also described in Stratego and format checkers are generated from their specifications.

In the next sections we will discuss various aspects of the specification of the warm fusion transformation system. In Section 1.5 we discuss the specification of the abstract syntax, checking subsets of an abstract syntax, and the specification of bound variable renaming and substitution by instantiating generic language independent algorithms. In Section 1.6 we present the overall structure of the transformer. In Section 1.7 we discuss the details of some of the transformations.

1.5 Abstract Syntax

The warm fusion transformation is performed on the abstract syntax of kernel Haskell, or `AHaskell`. The signature of the language is shown in Figure 1.4. It is a standard functional language with abstraction, application, data type deconstruction by means of case expressions, and a recursive let binding. The language is explicitly typed, which entails that types of variables in bindings can be declared, and that atomic expressions (variables, constructors and constants) can be annotated with their types. Polymorphic expressions are constructed by means of type abstraction and instantiated by means of type application. A program consists of a list of type and function definitions.

1.5.1 Format Checking

In the course of the transformation we encounter three intermediate formats that are subsets of `AHaskell` (Figure 1.4). The input format `hs-input` allows atomic expressions without type annotations because requiring annotations would clutter the source code. It also allows infix operators as syntactic sugar for prefix application. In the intermediate format `hs-typed` all atomic expressions are annotated with their types and are type correct. In addition, all operators are in prefix form. Like `hs-typed`, the output format `hs-output` requires fully annotated atoms, but it also allows expressions constructed using the `Build` and `Cata` operators. The latter are not allowed in the input to the transformation.

These three expression formats could be described by introducing three separate signatures with different constructors. This would, however, require three sets of names for the same constructs and trivial translations from one set to the next. Instead, we use one signature and the recursive patterns of Section 1.3 to characterize the three restrictions. These recursive patterns document the formats and can be used to check the inputs to the transformation components.

We now consider in turn the forms of expressions in each of the three subformats of `AHaskell`. Atomic expressions in the `hs-typed` format consist of a variable, constructor or literal and a type annotation as described by the patterns

```
AExp = Var(id) + Constr(id) + Lit(id)
atom(t) = Typed(AExp, t)
TypedVar = Typed(Var(id),Type)
TypedAtom = atom(Type)
```

where `Type` is a recursive pattern which describes the structure of `AHaskell`'s types. Type annotations are represented by means of the constructor `Typed`, which represents the `e::t` notation in Haskell. Note that these patterns are

parameterized with the format for types `t`. The basic shape of a `hs-typed` expression is described by the patterns

```
exp(e, t, pat, var) =
  Abs(list(var), option(t), e) +
  Case(e, list(alt(e, t, pat))) +
  Let(list(decl(e, t)), e) +
  App(e, list(e)) +
  TAbs(list(TVar(id)), e) +
  TInst(e, list(t))

alt(e, t, pat) =
  Alt(pat, option(t), e)

simple-pattern(var) =
  Constr(id) +
  App(Constr(id), list(var))

TypedPat =
  simple-pattern(TypedVar)
```

and a typed expression is characterized by the recursive pattern

```
TypedExp =
  rec e(TypedAtom + exp(e, Type, TypedPat, TypedVar))
```

In the `hs-input` format, atomic expressions (variables, constructors and literals) can be untyped. Furthermore, infix operator applications in addition to prefix application and binary in addition to n-ary application are allowed. This is described by

```
PreVar =
  Var(id) +
  Typed(Var(id), PreType)

PrePat =
  simple-pattern(PreVar)
  + rec x(AppBin(x, PreVar) + Constr(id))

pre-exp(e) =
  OpApp(e, id, e) +
  AppBin(e, e) +
  Negation(e) +
  If(e, e, e)

PreExp =
  rec e(AExp + atom(PreType) + pre-exp(e) +
        exp(e, PreType, PrePat, PreVar))
```

The typechecker normalizes infix and binary applications to n-ary applications and annotates all atomic expressions with their types.

Finally, the expressions in the output format `hs-output` are typed expressions extended with `Build` and `Cata` operators:

```
ext-exp(e, t) =
  Cata(t, t, list(e)) +
  Build(t, e)

ExtExp =
  rec e(TypedAtom + exp(e, Type, TypedPat, TypedVar) +
        ext-exp(e, Type))
```

1.5.2 Variable Renaming and Substitution

AHaskell has variable binding constructs. The Stratego library defines (using standard Stratego) the generic, language independent strategies `rename` for renaming bound variables, `substitute` for parallel substitution of expressions for variables, and `free-vars` for the extraction of the free variables from an expression. These operations are instantiated by declaring the shape of variables, indicating the binding constructs, and identifying the binding positions. We illustrate their instantiation for `AHaskell`. The implementation of the generic algorithms is presented in [34].

The following rules are used to describe the shape of variables.

```
IsVar(s)  : Var(x) -> Var(<s> Var(x))
ExpVar    : Typed(Var(x),_) -> Var(x)
ExpVar    : Var(x) -> Var(x)
ExpVars   : Var(x) -> [Var(x)]
```

The binding constructs of expressions are lambda abstraction, case alternatives, and let binding. The rules `ExpBnd` define the projection from these constructs to the list of variables that they bind.

```
ExpBnd  : Abs(xs, _, _) -> <map(ExpVar)> xs
ExpBnd  : Alt(App(c, xs), t, e) -> <map(ExpVar)> xs
ExpBnd  : Let(decls, e) -> <filter(DeclVar)> decls
DeclVar : Valdef(Var(x), e) -> Var(x)
```

Using the rules above the instantiations of `free-vars`, `substitute`, and `rename` for expressions are

```
expvars   = free-vars(ExpVars, ExpBnd)
exprename = rename(IsVar, ExpBnd)
expsubst  = substitute(Typed(Var(id),id) + Var(id), etrename)
```

Proper substitution entails that bound type variables in expressions that are substituted for term variables are also renamed, and so an exercise similar to that above must be carried out for type variables. This gives rise to the corresponding operators `tpvars`, `tpsubst`, and `tprename` for types. The strategy `etrename` is the sequential composition of `exprename` and `tprename`.

1.6 Transformer: Big Picture

In this section we discuss the specification of the top-level of the warm fusion transformer. The reader is directed to the next chapters, from which the following code is excerpted, for a complete code listing.

1.6.1 Transforming a Program

The main strategy takes a program, i.e., a list of type and function declarations, and transforms each in turn. This is achieved by a transition step for each declaration:

```
Main = etrename;
       where(collect-data-defs);
       InitWF;
       repeat(TransformDecl <+ NormD);
       ExitWF
```

Note that all bound variables in the entire program are first renamed to establish the unique variable invariant. Furthermore, the strategy `collect-data-defs` finds the data type definitions in the program and stores them in a symbol table for later reference. The initial configuration is created from a list of declarations and the final configuration derives a transformed list of declarations:

```
InitWF :
  ds -> ([], [], ds)
ExitWF :
  (ds1, ds2, []) -> <reverse> ds2
```

The first accumulator list stores the functions that have been transformed to build-cata form. These are used for inlining in other functions. The second accumulator list stores all functions, including the non-transformed ones.

A definition is transformed by first inlining functions that were transformed earlier (in the list `ds1`) and then applying the warm fusion transformation to it.

```
TransformDecl :
  (ds1, ds2, [d | ds3]) -> ([d' | ds1], [d' | ds2], ds3)
  where <ior(inline(!ds1), Transform)> d => d'
```

Inlining and transformation can fail. If at least one succeeds then the result is considered to be transformed and is added to both accumulator lists. (The rule `ior` computes the inclusive or of two strategies, i.e., `ior(s1,s2)` applies `s1`, `s2` or both.) If both fail then the function is added only to the list of non-transformed functions using the rule

```
NormD :
  (ds1, ds2, [d| ds3]) -> (ds1, [d| ds2], ds3)
```

Inlining is achieved by replacing calls to functions in a given list of declarations by (renamings of) their bodies and then simplifying the resulting expressions using the rules of Section 1.7. Inlining replaces as many calls as possible, but at least one call must be replaced in order for it to succeed:

```
inline(mkenv) = manytd(Inline(mkenv)); simplify
```

The function to be inlined is looked up in the list of declarations passed to the rule `Inline`. The strategy `<not(in)>` checks for recursion in the definition of the function. Recursive functions are not inlined.

```
Inline(mkenv) :
  Typed(Var(x), t) -> <tpsubst; etrename> (sbs, e)
  where mkenv; fetch(?Valdef(Var(x), e)); <not(in)> (Var(x), e);
        <tpunify> [(<type> e, t)] => sbs
```

1.6.2 Transforming a Definition

The basic algorithm for transforming a recursive definition to build-cata form — as defined in [16] and illustrated in Section 1.2 — is the following:

```
Transform =
  IntroBuildCata;
  simplify;
  SplitBodyCP;
  Unfold1in2;
  [id, simplify;
       MakeCataBody];
  Unfold2in1;
  simplify
```

This strategy introduces the build-cata identity, splits the body into a wrapper and a worker, unfolds the wrapper in the worker, transforms the worker into a catamorphism, and unfolds the worker back in the wrapper. In between it simplifies the definitions.

As we remarked in Section 1.2 this procedure applies only to functions that both consume and produce data structures. To accommodate functions that either only consume or only produce data structures we refine the algorithm using the same building blocks to the following:

```
Transform =
  ((IntroBuildCata;
    simplify;
    (ConsumerProducer
     <+ Producer
     <+ NonRecursiveProducer))
   <+ Consumer);
  simplify
```

The strategies `ConsumerProducer`, `Consumer`, `Producer`, and `NonRecursive-Producer` represent the different possible ways of transforming a function. The strategy `Consumer` is applied when introduction of the outer `build` and `cata` fails. In this case the output type of the function is not a data type and so the function does not produce a data structure. It may, however, still be a consumer. If, on the other hand, the introduction of the outer `build` and `cata` succeeds, then `ConsumerProducer` splits the body of the function into a wrapper and a worker and tries to derive a catamorphism for the worker. If deriving a catamorphism from the worker fails, then the function is only a producer.

Although it is not apparent at this level of abstraction, the introduction of the outer `build` and `cata` is governed by the input and output types of the function being transformed. We consider the details of the above transformation in Section 1.7.

The derivation of a catamorphism for the worker and unfolding it back in the wrapper is defined in the strategy `BodyToCata`:

```
BodyToCata =
  Unfold1in2;
  [id, simplify;
```

```
        SplitBodyP;
        Unfold1in2;
        [id, simplify;
             MakeCataBody];
        Unfold2in1];
    Unfold2in1
```

Unlike `Transform`, this strategy splits and unfolds the worker twice in order to recognize the abstracted constructors as static parameters.

1.7 Transformer: Details

In this section we go into the details of some of the transformations mentioned above.

1.7.1 Simplification

The simplifier consists of a number of standard simplification rules for functional programs such as beta reduction:

```
BetaOne :
  App(Abs([x|xs], t, e), [a|as]) ->
  App(Abs(xs, t, <expsubst> ([x], [a], e)), as)
  where <value> a + <linear> (x, e)
```

Here, `value` and `linear` are strategies that prevent duplication of work during reduction. An expression is a value if it represents either a function or a data object; a variable `v` appears linearly in the expression `b` if reduction of `b` can never cause duplication of any term substituted for `v`. Terms which do not encode computation are literally copied regardless of whether or not the variables they instantiate occur linearly in their host terms.

The beta reduction rule `BetaOne` reduces an application of a function to its first argument. The following rule reduces such an application as far as possible, either exhausting all formal or all actual parameters.

```
Beta :
  App(Abs(xs, t, e), as) ->
  App(Abs(ys, t, <expsubst> (sbs, e)), bs)
  where <rest-zip(id)> (xs, as) => (ys, bs, sbs);
        (<lzip((id,value) + (Fst,id); linear)> (sbs, e))
```

Other simplification rules include elimination of dead `let` bindings, inlining of let bindings, case specialization, distribution of application over cases, uncurrying of expression and type applications; see the definition of `basic-rules` below. A particularly important rule for the warm fusion transformation is, of course, cata-build fusion:

```
CataBuild :
  App(Cata(t1, t2, fs), [Build(t1, g)]) ->
  App(TInst(g, [t2]), fs)
```

Here `t1` is the input type for the catamorphism and `t2` is its return type. Similarly, `t1` is the type of `build`'s output.

These basic rules can be combined in various ways to build simplifiers, depending on the desired effect. We use the following configuration in the warm fusion transformer.

```
basic_rules =
  Beta + Eta + (Inl; Dead) + TEta + TBeta +
  CaseConstr + CaseDistL + CaseDistR + Uncurry

basic-cata = CataConstr + CataBuild + basic_rules

simplify = innermost(basic-cata)
```

The strategy `innermost` is defined by

```
innermost(s) = rec x(all(x); (s; x <+ id))
```

Although the definition of `simplify` here uses innermost reduction, Stratego's separation of logic from control make it particularly convenient to change the term reduction strategy used in the simplifier.

1.7.2 Build-Cata Introduction

The initial `build-cata` identity is introduced into the body of the function definition under its leading abstractions:

```
IntroBuildCata = Valdef(id, under-abs(MkBuildCata))
```

where the notion 'under its leading abstractions' can be expressed by the recursive pattern

```
under-abs(s) = rec x((Abs(id, id, x) + TAbs(id, x)) <+ s)
```

In concrete syntax the `build-cata` identity has the form

```
build[t1](/\t2 -> \fs :: t2 -> (cata[t1][t2](fs) e))
```

where `t1` is the type of the expression `e`, `t2` is a new type variable and the `fs` are the abstract constructors corresponding to the constructors of the data type. Generation of this form is defined by the following rule:

```
MkBuildCata :
  e -> Build(t1, TAbs([t2], Abs(fs, Some(t2),
                                App(Cata(t1, t2, fs), [e]))))
  where new-tvar => t2; <type> e => t1;
        <get-constructors> t1 => cdecls;
        <lzip(AbsConstr)> (cdecls, (t1, t2)) => fs
```

Type information plays a crucial role in `build-cata` introduction and subsequent processing. It is used not only to determine which instances of the `Cata` and `Build` functions to introduce, but also to generate arguments of the appropriate types for these instances. The strategy `type` derives the type from an expression. The strategy `get-constructors` obtains the constructor declarations corresponding to the type of `e`. For each constructor of the data type an abstract constructor (variable) with the appropriate type is constructed by rule `AbsConstr`:

```
AbsConstr :
  (ConstrDecl(_, _, c, ts), (t1, t2)) ->
  Typed(Var(f), TFun(ts', t2))
  where new => f; <map(try(?t1;!t2))> ts => ts'
```

The rule creates a variable expression with new variable `f` and its type. The function has the same number of arguments as the original constructor. The output of the function is of type `t2`. Where the constructor has a recursive argument, indicated by the recursion type `t1`, the output type `t2` is instantiated. The other arguments remain the same type.

1.7.3 Splitting Function Definitions

Splitting a function into a wrapper and a worker involves determining where in the body the split is performed, which variables the worker is abstracted over, creating the definition of the worker and replacing the expression in the wrapper body by a call to the worker. There are several ways to do this. We discuss one of them.

The strategy `SplitBodyP` first computes the non-static parameters `vs` of the function definition and then splits the body. This is achieved by instantiating `SplitBody` with a strategy for splitting expressions:

```
SplitBodyP =
  where(NonStaticParams => vs);
  SplitBody(SplitExpr(!vs))
```

`NonstaticParams` extracts the nonstatic parameters from a function definition; the function's case selector must be the head of the list of nonstatic parameters in order to satisfy the strictness requirement of the promotion theorem. Given any list `xs` of value and type variables, the rule `SplitExpr` creates a definition for a function with a new name `f` that has the expression as its body and abstracts over `xs`. It also creates a call to `f` with `xs` as arguments. The definition of `SplitExpr` assumes that the type parameters to a function are always static.

```
SplitExpr(mkxs) :
  e -> (App(Typed(Var(f), t), xs), Valdef(Var(f), body))
  where mkxs => xs; new => f;
        <etrename> Abs(xs, Some(<type> e), e) => body;
        <type> body => t
```

Given a strategy `split` for splitting an expression, rule `SplitBody` splits the body of a function definition by creeping under its leading abstractions and splitting the expression it encounters there.

```
SplitBody(split) :
  Valdef(Var(x), body) -> [Valdef(Var(x), body'), def]
  where <under-abs-build(split => (e, def); !e)> body => body'
```

The split results in an expression (the call) and a new definition. The expression `split => (e, def); !e` matches the result of splitting against the pattern `(e, def)` and then replaces it by just the expression. The binding to `def` is used in the right-hand side of the rule, where a list of two definitions is created.

Since we want to split off the worker under the build expression, if present, we use a variant of the `under-abs` pattern that we saw before.

```
under-abs-build(split) =
  rec x((Abs(id,id,x) + TAbs(id,x) + Build(id,split)) <+ split)
```

Similar patterns can be used to describe other contexts in which a transformation has to take place.

Parameterizing over `under-abs-build` as well as `split` would make `SplitBody` a completely generic splitting strategy. However, even as defined here, `SplitBody` is a general strategy for splitting under any type and term abstractions and any `build`s in a function definition. Our splitting mechanism therefore generalizes that from [24] upon which the wrapper-worker decomposition in [16] is based. The extra generality is useful: splitting a function definition into a wrapper and a worker sometimes requires splitting under a function's leading `build`, while at other times no `build`s are present. The strategy `under-abs-build` given here is general enough to accommodate both situations.

1.7.4 Unfolding

Unfolding is defined by the following contextual rules [36] that replace all occurences of atoms with the name of the function being unfolded by its body.

```
Unfold1in2 :
   [Valdef(Var(x),body1), Valdef(Var(y),body2[Typed(Var(x),_)])]
-> [Valdef(Var(x),body1), Valdef(Var(y),body2[body1'](alltd))]
   where <exprename> body1 => body1'

Unfold2in1 :
   [Valdef(Var(x),body1[Typed(Var(y),_)]), Valdef(Var(y),body2)]
-> Valdef(Var(x),body1[body2'](alltd))
   where <not(in)> (Var(y), body2); <exprename> body2 => body2'
```

1.7.5 Cata Promotion

In Section 1.2 we discussed how a catamorphism can be derived from a recursive definition using the promotion theorem. The core of the promotion is the creation of a function

```
h = \z1 ... zn -> e(c(y1)...(yn))
```

for each constructor `c` with `n` arguments. The function `e` is then unfolded exactly once, and the result is simplified using the standard rules, together with a dynamically generated set of rules that rewrite recursive applications involving the `y`is to the appropriate variables `z`i. The abstract syntax of the initial form of the function `h` is

```
Abs(zs, App(e, [App(Typed(Constr(c), TFun(ts, t)), ys)]))
```

The rule `DynRules` creates for a specific constructor, the lists of `y` and `z` variables and the corresponding dynamic rewrite rules. The strategy `dsimplify` extends the normal simplification with the application of these dynamic rules.

```
dsimplify(mkrls) = innermost(AppDynRule(mkrls) <+ basic_rules)
```

Putting this together the rule `MkH` creates the replacement function corresponding to a constructor of the original function's input data type.

```
MkH :
  (ConstrDecl(_, _, c, ts), (g, e, t)) -> h
  where
    <DynRules> (t, g, c) => (ys, zs, rls);
    !Typed(Constr(c), TFun(ts, t)) => ct;
    <dsimplify(!rls)>
       Abs(zs, None, App(<etrename> e,[App(ct, ys)])) => h;
    <not(oncetd({y : ?Var(y);
                 where(<fetch(Typed(Var(?y),id))> ys)}))> h
```

Note that the bound variables in expression `e` are renamed to maintain the unique variable invariant.

These replacement functions are then used by `MakeCataBody` to construct the catamorphic version of that function's worker. Unfolding the worker in the wrapper yields the build-cata form of the function definition being transformed.

```
MakeCataBody :
  Valdef(Var(g), e) -> Valdef(Var(g), Cata(t1, t2, hs))
  where <type> e => tg;
        <split(dom, range)> tg => (t1, t2);
        <get-constructors> t1 => cdecls;
        <lzip(MkH)> (cdecls, (Typed(Var(g), tg), e, t1)) => hs
```

This concludes our sample of the specification. The complete text of the specification can be found in the next chapters.

1.8 Related Work

The first ideas for rewriting strategy operators with general traversal operators are described in [17]. In [36] these ideas are formalized by means of an operational semantics and are extended to the full set of System S operators by splitting simple rewrite rules into match, build and scope. This allows easy expression of contextual rules. An application to the specifiation of optimizers is discussed. In [35] it is shown how System S can be used to describe various features and evaluation strategies of traditional conditional rewriting systems. In [32] three programming idioms for *strategic pattern matching* are studied: recursive patterns, contextual rules, and overlays. The implementation of generic algorithms such as used for variable renaming and substitution is discussed in [34]. For a discussion of related work on rewriting strategies see [35]. The relation to other systems for program transformation is discussed in [36].

Techniques for program fusion can be classified into two broad categories: *search-based* and *calculation-based*. The earliest techniques for program fusion [6, 29, 37, 7] were search-based. These rely on analyses of the *fold-unfold* transformation process of Burstall and Darlington to fuse compositions of recursive functions. In search-based fusion it is necessary to keep track at each step of the transformation process of all function calls that have been made. New function definitions to be used in unfolding must then be introduced. Search-based fusion is systematic, but relies on clever control mechanisms to avoid the possibility of infinite sequences of transformations by repeated unfolding of function definitions. As a result, good implementations of search-based fusion techniques have been somewhat difficult to achieve.

The warm fusion method and the short cut to deforestation which it facilitates are in the more recent tradition of calculation-based fusion [26, 11, 27, 16, 13]. In calculation-based fusion the recursive structure of each component participating in the fusion is made explicit. This enables fusion by direct application of simple transformation laws like the cata-build rule and the acid rain theorem [27]. The theoretical basis for calculation-based fusion lies in the study of *constructive algorithmics* [8, 19, 20].

1.9 Future Work

The implementation of program fusion algorithms offers many additional opportunities for investigation. Among the issues pertaining directly to the Stratego implementation and meriting attention are: experimenting with various orders and strategies for applying the simplification transformations; experimenting with more unfolding of function definitions when converting recursion to catamorphisms via fold promotion so that fusion is not unnecessarily blocked; making inlining more context sensitive, so that build-cata forms are inlined only when there is the possibility of fusion via the short cut; and extending the transformations with Gill's `augment`. Benchmarking to determine the sense(s) in which deforested programs are "better" than their monolithic counterparts is also appropriate for the current warm fusion implementation. So is comparison of the Stratego specification with other implementations of warm fusion.

Other lines of inquiry involve the integration of automatic fusion tools into existing systems. Candidate systems include the optimizer of the RML compiler discussed in [28, 36], as well as state-of-the-art functional language compilers. Nemeth [21] has recently implemented warm fusion in the Glasgow Haskell Compiler and reported benchmarks on programs from the nofib suite [22].

Finally, rather than using Stratego as a tool to help deepen our understanding of program fusion techniques, we can turn the relationship between strategy-based languages and program fusion on its head and ask about possible applications of fusion to strategy-based languages. Can we formalize our intuition that certain combinations of strategies should themselves be amenable to suitable forms of strategy fusion? Is it possible, for example, to make precise the observation that

```
!C(t1,...,tn); ?C(t1',...,tn') = !t1;?t1';...;!tn;?tn'
```

assuming that the term that is built is not used again?

1.10 Conclusion

We have presented a case study of the application of Stratego to build a complete, non-trivial program transformation system. Table 1.1 shows the sizes of the main components of the transformation system in number of modules, lines of code (text including comments), number of rules and number of strategies. Note that these figures do not include the signature and the pretty-printing modules. Distributed over time, it took us about 30 days to develop the entire transformation tool from scratch including a syntax definition for full Haskell. The development time included finding out how to program in Stratego and

language	component	mod	LOC	cons	rules	strat
SDF	Haskell.sdf		650		300	–
Stratego	Warm Fusion	11	739	0	60	31
Stratego	Format checking	1	202	1	1	32
Stratego	Haskell Library	4	246	0	32	21
Stratego	Haskell Normalize	1	75	0	17	3
Stratego	Haskell Typecheck	1	120	1	13	6
Stratego	Subtotal Specification	18	1382	2	123	93
Stratego	Signature	28	544	103	0	0
Stratego	Pretty-Printer	28	671	0	90	7
Stratego	Total Specification	74	2597	105	213	100
Stratego	Stratego Library	48	3634	65	131	317

Table 1.1: Size metrics of main components of the specification. Measuring number of modules (mod), lines of code (LOC) including documentation, number of constructors (cons), rules and strategies (strat).

developing programming idioms. That is, when undertaking this case study, Stratego was a new language, even for its author, and discovering idioms of use beyond the basic paradigm takes time. The development was aided by the wealth of generic, language independent rules and strategies in the Stratego library [33].

This case study strengthens our view that rewriting strategies are a good paradigm for the implementation of program transformation systems. The specification is highly modular at all levels and can easily be modified or extended with new transformations. It will serve as the basic infrastructure for further experimentation with transformations on full Haskell. The specification also provides examples of several Stratego idioms that can be used in the implemention of transformation systems for other languages. In particular the specification shows the use of compound rules, recursive patterns, distributed patterns, exchange of information between transformation rules through parameterized strategies and the compact specification of variable renaming, substitution, and free variable projection.

Acknowledgements The authors would like to thank various anonymous referees for their comments on earlier versions of this paper.

Chapter 2

Examples

This chapter presents the verbatim input and output of the warm fusion transformation tool for several small programs. The output is given in two versions. In the first the type annotations have been stripped of expressions to provide a readable program. The second is the result of the transformation with all type annotation present.

2.1 Lists

2.1.1 Input

```
module SOS where {

-- Booleans

data Bool = False | True;

-- Integers

(*)  :: Int -> Int -> Int;
(+)  :: Int -> Int -> Int;
(>)  :: Int -> Int -> Bool;
(==) :: Int -> Int -> Bool;

square :: Int -> Int;
square = \x -> (x * x);

-- Lists

data List a = Nil | Cons a (List a);

map :: (a -> b) -> List a -> List b;
map = \f l ->
      case l of
      { Nil       -> Nil
      ; Cons x xs -> Cons(f x)(map f xs)
```

```
      };

foldr :: b -> (a -> b -> b) -> List a -> b;
foldr = \n c xs ->
        case xs of
        { Nil       -> n
        ; Cons y ys -> c y (foldr n c ys)
        };

upto :: Int -> Int -> List Int;
upto = \low high ->
       case low > high of
       { False -> Cons low (upto(low + 1)(high))
       ; True  -> Nil
       };

sum :: List Int -> Int;
sum = foldr 0 (+);

sos :: Int -> Int -> Int;
sos = \lo hi ->
      sum(map(square)(upto lo hi))

}
```

2.1.2 Output

```
module SOS where {

data Bool = False
          | True;

(*) :: (Int) -> (Int) -> Int;

(+) :: (Int) -> (Int) -> Int;

(>) :: (Int) -> (Int) -> Bool;

(==) :: (Int) -> (Int) -> Bool;

square :: (Int) -> Int;

square = (\c_0 ->
          (c_0 * c_0));

data List a = Nil
            | Cons (a) (List a);

map :: ((a) -> b) -> (List a) -> List b;
```

```
map = (\d_0 e_0 ->
       build(List b_0,
             /\f_1 ->
             (\e_2 f_2 ->
              cata[List a_0][f_1](e_2, (\g_2 h_2 ->
                                         f_2(d_0(g_2))(h_2)))(e_0))));

foldr :: (b) -> ((a) -> (b) -> b) -> (List a) -> b;

foldr = (\j_0 k_0 l_0 ->
         cata[List g_0][h_0](j_0, k_0)(l_0));

upto :: (Int) -> (Int) -> List Int;

upto = (\o_0 p_0 ->
        build(List Int,
              /\n_4 ->
              (\i_4 j_4 ->
               let { k_4 = (\l_4 ->
                            case (l_4 > p_0) of
                            { False -> j_4(l_4)(k_4((l_4 + 1)))
                            ; True -> i_4
                            }) }
                in k_4(o_0))));

sum :: (List Int) -> Int;

sum = cata[List Int][Int](0, (+));

sos :: (Int) -> (Int) -> Int;

sos = (\q_0 r_0 ->
       let { q_5 = (\r_5 ->
                    case (r_5 > r_0) of
                    { False -> (square(r_5) + q_5((r_5 + 1)))
                    ; True -> 0
                    }) }
        in q_5(q_0))

}
```

2.1.3 Output (Fully Typed)

```
module SOS where {

data Bool = False
          | True;

(*) :: (Int) -> (Int) -> Int;
```

```
(+) :: (Int) -> (Int) -> Int;

(>) :: (Int) -> (Int) -> Bool;

(==) :: (Int) -> (Int) -> Bool;

square :: (Int) -> Int;

square = (\(c_0 :: Int) :: Int ->
          ((*) :: (Int) -> (Int) -> Int)((c_0 :: Int))((c_0 :: Int)));

data List a = Nil
            | Cons (a) (List a);

map :: ((a) -> b) -> (List a) -> List b;

map = (\(d_0 :: (a_0) -> b_0) (e_0 :: List a_0) :: List b_0 ->
       build(List b_0,
             /\f_1 ->
             (\(e_2 :: f_1) (f_2 :: (b_0) -> (f_1) -> f_1) :: f_1 ->
              cata[List a_0][f_1]
                ((e_2 :: f_1),
                 (\(g_2 :: a_0) (h_2 :: f_1) ->
                  (f_2 :: (b_0) -> (f_1) -> f_1)
                    ((d_0 :: (a_0) -> b_0)((g_2 :: a_0)))
                    ((h_2 :: f_1))))((e_0 :: List a_0)))));

foldr :: (b) -> ((a) -> (b) -> b) -> (List a) -> b;

foldr = (\(j_0 :: h_0) (k_0 :: (g_0) -> (h_0) -> h_0) (l_0 :: List g_0) :: h_0 ->
         cata[List g_0][h_0]((j_0 :: h_0), (k_0 :: (g_0) -> (h_0) -> h_0))
           ((l_0 :: List g_0)));

upto :: (Int) -> (Int) -> List Int;

upto = (\(o_0 :: Int) (p_0 :: Int) :: List Int ->
        build(List Int,
              /\n_4 ->
              (\(i_4 :: n_4) (j_4 :: (Int) -> (n_4) -> n_4) :: n_4 ->
               let { k_4 = (\(l_4 :: Int) :: n_4 ->
                            case ((>) :: (Int) -> (Int) -> Bool)
                                   ((l_4 :: Int))
                                   ((p_0 :: Int)) of
                            { False :: Bool
                              -> (j_4 :: (Int) -> (n_4) -> n_4)
                                   ((l_4 :: Int))
                                   ((k_4 :: (Int) -> n_4)
                                      (((+) :: (Int) -> (Int) -> Int)
                                         ((l_4 :: Int))
```

```
                                        ((1 :: Int))))
                            ; True :: Bool -> (i_4 :: n_4)
                            }) }
                in (k_4 :: (Int) -> n_4)
                 ((o_0 :: Int)))));

sum :: (List Int) -> Int;

sum = cata[List Int][Int]((0 :: Int), ((+) :: (Int) -> (Int) -> Int));

sos :: (Int) -> (Int) -> Int;

sos = (\(q_0 :: Int) (r_0 :: Int) :: Int ->
       let { q_5 = (\(r_5 :: Int) :: Int ->
                     case ((>) :: (Int) -> (Int) -> Bool)
                            ((r_5 :: Int))
                            ((r_0 :: Int)) of
                     { False :: Bool -> ((+) :: (Int) -> (Int) -> Int)
                                        ((square :: (Int) -> Int)
                                           ((r_5 :: Int)))
                                        ((q_5 :: (Int) -> Int)
                                           (((+) :: (Int) -> (Int) -> Int)
                                              ((r_5 :: Int))
                                              ((1 :: Int))))
                     ; True :: Bool -> (0 :: Int)
                     }) }
        in (q_5 :: (Int) -> Int)
         ((q_0 :: Int)))

}
```

2.2 Pairs

2.2.1 Input

```
module Pairs where {

(+) :: Int -> Int -> Int;

data Pair a b = Pair a b;

id :: a -> a;
id = \ x -> x;

inc :: Int -> Int;
inc = \ x -> (x + 1);

swap :: Pair a b -> Pair b a;
```

```
swap = \ p -> case p of { Pair x y -> Pair y x };

cross :: (a -> c) -> (b -> d) -> Pair a b -> Pair c d;
cross = \ f g p -> case p of { Pair x y -> Pair (f x) (g y) };

split :: (a -> b) -> (a -> c) -> a -> Pair b c;
split = \ f g x -> Pair (f x) (g x);

add :: Pair Int Int -> Int;
add = \ p -> case p of { Pair i j -> i + j };

swapadd :: Pair Int Int -> Int;
swapadd = \ p -> add(swap(p));

test1 :: Int -> Int;
test1 = \ x -> add(swap(split inc inc x));

}
```

2.2.2 Output

```
module Pairs where {

(+) :: (Int) -> (Int) -> Int;

data Pair a b = Pair (a) (b);

id :: (a) -> a;

id = (\d_0 ->
      d_0);

inc :: (Int) -> Int;

inc = (\e_0 ->
       (e_0 + 1));

swap :: (Pair a b) -> Pair b a;

swap = (\j_0 ->
        build(Pair b_0 c_0,
              /\q_1 ->
              (\k_2 ->
               cata[Pair c_0 b_0][q_1]((\l_2 m_2 ->
                                        k_2(m_2)(l_2)))(j_0))));

cross :: ((a) -> c) -> ((b) -> d) -> (Pair a b) -> Pair c d;

cross = (\p_0 q_0 r_0 ->
```

```
         build(Pair h_0 i_0,
               /\x_2 ->
               (\t_3 ->
                cata[Pair f_0 g_0][x_2]((\u_3 v_3 ->
                                         t_3(p_0(u_3))(q_0(v_3))))(r_0))));

split :: ((a) -> b) -> ((a) -> c) -> (a) -> Pair b c;

split = (\u_0 v_0 w_0 ->
         build(Pair m_0 n_0,
               /\w_3 ->
               (\x_3 ->
                x_3(u_0(w_0))(v_0(w_0)))));

add :: (Pair Int Int) -> Int;

add = cata[Pair Int Int][Int]((+));

swapadd :: (Pair Int Int) -> Int;

swapadd = cata[Pair Int Int][Int]((\f_5 g_5 ->
                                   (g_5 + f_5)));

test1 :: (Int) -> Int;

test1 = (\b_1 ->
         (inc(b_1) + inc(b_1)))

}
```

2.2.3 Output (Fully Typed)

```
module Pairs where {

(+) :: (Int) -> (Int) -> Int;

data Pair a b = Pair (a) (b);

id :: (a) -> a;

id = (\(d_0 :: a_0) :: a_0 ->
      (d_0 :: a_0));

inc :: (Int) -> Int;

inc = (\(e_0 :: Int) :: Int ->
       ((+) :: (Int) -> (Int) -> Int)((e_0 :: Int))((1 :: Int)));

swap :: (Pair a b) -> Pair b a;
```

```
swap = (\(j_0 :: Pair c_0 b_0) :: Pair b_0 c_0 ->
        build(Pair b_0 c_0,
              /\q_1 ->
              (\(k_2 :: (b_0) -> (c_0) -> q_1) :: q_1 ->
               cata[Pair c_0 b_0][q_1]((\(l_2 :: c_0) (m_2 :: b_0) ->
                                        (k_2 :: (b_0) -> (c_0) -> q_1)
                                          ((m_2 :: b_0))
                                          ((l_2 :: c_0))))
                 ((j_0 :: Pair c_0 b_0)))));

cross :: ((a) -> c) -> ((b) -> d) -> (Pair a b) -> Pair c d;

cross = (\(p_0 :: (f_0) -> h_0) (q_0 :: (g_0) -> i_0) (r_0 :: Pair f_0 g_0) :: Pair h_0 i_0 -
         build(Pair h_0 i_0,
               /\x_2 ->
               (\(t_3 :: (h_0) -> (i_0) -> x_2) :: x_2 ->
                cata[Pair f_0 g_0][x_2]
                  ((\(u_3 :: f_0) (v_3 :: g_0) ->
                    (t_3 :: (h_0) -> (i_0) -> x_2)
                      ((p_0 :: (f_0) -> h_0)((u_3 :: f_0)))
                      ((q_0 :: (g_0) -> i_0)((v_3 :: g_0)))))
                  ((r_0 :: Pair f_0 g_0)))));

split :: ((a) -> b) -> ((a) -> c) -> (a) -> Pair b c;

split = (\(u_0 :: (l_0) -> m_0) (v_0 :: (l_0) -> n_0) (w_0 :: l_0) :: Pair m_0 n_0 ->
         build(Pair m_0 n_0,
               /\w_3 ->
               (\(x_3 :: (m_0) -> (n_0) -> w_3) :: w_3 ->
                (x_3 :: (m_0) -> (n_0) -> w_3)
                  ((u_0 :: (l_0) -> m_0)((w_0 :: l_0)))
                  ((v_0 :: (l_0) -> n_0)((w_0 :: l_0))))));

add :: (Pair Int Int) -> Int;

add = cata[Pair Int Int][Int](((+) :: (Int) -> (Int) -> Int));

swapadd :: (Pair Int Int) -> Int;

swapadd = cata[Pair Int Int][Int]((\(f_5 :: Int) (g_5 :: Int) ->
                                   ((+) :: (Int) -> (Int) -> Int)
                                     ((g_5 :: Int))
                                     ((f_5 :: Int))));

test1 :: (Int) -> Int;

test1 = (\(b_1 :: Int) :: Int ->
         ((+) :: (Int) -> (Int) -> Int)
           ((inc :: (Int) -> Int)((b_1 :: Int)))
```

```
            ((inc :: (Int) -> Int)((b_1 :: Int))))

}
```

Chapter 3

Concrete Syntax

3.1 Syntax Definition in SDF2

This chapter presents a definition of the syntax of a subset of the functional programming language Haskell in SDF2, a formalism for syntax definition [31]. From the syntax definition a generalized-LR parser is generated by the `pgen` program [30]. The parser produces parse trees in the AsFix format, which is represented using ATerms. Parse trees are transformed into abstract syntax trees by means of the generic `implode-asfix` program. This program uses the constructor annotations `cons("...")` in the grammar productions to translate parse tree nodes into abstract syntax tree nodes. For more information about SDF2 see [2].

3.2 Haskell in SDF2

The subset that is covered by the grammar can be characterized as Core Haskell with types. All details of the lexical syntax are defined according to the standard [23]. Some syntactic sugar (e.g., `if`) is supported, but most is not. Not included in the subset are import-export declarations, list comprehensions, monad notation, type classes, records, and where clauses. The syntax definition presented here is part of a larger definition that covers the entire definition in the Haskell standard. Since those parts are not used in the subsequent transformation they are not included. One notable difference with the standard is that the offset rule is not supported. This entails that all semicolons and curly braces have to be supplied in the program.

```
module Main
imports Haskell-Kernel
exports
  sorts Module

module Haskell-Kernel
imports Haskell-Layout
        Haskell-Identifiers
        Haskell-Keywords
        Haskell-Identifier-Sorts
```

```
        Haskell-Numbers
        Haskell-Strings
        Haskell-Literals
        Haskell-Modules
        Haskell-Types
        Haskell-Type-Declarations
        Haskell-Signature-Declarations
        Haskell-Expressions
        Haskell-Case-Alternatives
        Haskell-Value-Definitions
        Haskell-Infix
        ExtraSepLists1
        ExtraSepLists
        ExtraSepLists0
```

3.3 Lists with Separators

Haskell allows at several points redundant separators in lists. Such lists are generically defined in the following parameterized modules.

```
module ExtraSepLists [Elt Sep List]
exports
  context-free syntax
    Elt          -> List {cons("Ins")}
    List Sep Elt -> List {cons("Snoc")}
    List Sep     -> List


module ExtraSepLists0 [Elt Sep List]
exports
  context-free syntax
    Elt          -> List {cons("Ins")}
    List Sep Elt -> List {cons("Snoc")}
    List Sep     -> List
                 -> List {cons("Nil")}
```

3.4 Lexical Syntax

```
module Haskell-Layout
exports
  lexical syntax
    WhiteChar                    -> LAYOUT
    Comment                      -> LAYOUT
    NComment                     -> LAYOUT

    [\ \t\n]                     -> WhiteChar

    "--" ~[\n]* [\n]             -> Comment
```

```
    "{-" {L-Char* NComment}* "-}" -> NComment

    ~[\-\{]                       -> L-Char
    Hyphen                        -> L-Char
    CurlyOpen                     -> L-Char
    [\-]                          -> Hyphen
    [\{]                          -> CurlyOpen
  lexical restrictions
    Hyphen    -/- [\}]
    CurlyOpen -/- [\-]
  context-free restrictions
    LAYOUT? -/- [\ \t\n] | [\-].[\-] | [\{].[\-]


module Haskell-Identifiers
exports
  lexical syntax
    [a-z][A-Za-z0-9\'\_]*       -> VARID
    [A-Z][A-Za-z0-9\'\_]*       -> CONID

    %% Question: underscore in identifiers according to standard???

    [\!\#\$\%\&\*\+\.\/]
    \/ [\<\=\>\?\@\\\^\|\-\~] -> Symbol
    Symbol (Symbol | [\:])*   -> VARSYM
    [\:] (Symbol | [\:])*     -> CONSYM
    ReservedOp                -> VARSYM {reject}
    ReservedOp                -> CONSYM {reject}

  lexical restrictions
    CONID VARID -/- [A-Za-z0-9]
    VARSYM      -/- [\!\#\$\%\&\*\+\.\/] \/ [\<\=\>\?\@\\\^\|\-\~]

   context-free syntax
    Modid "." VARID           -> QVARID  {cons("QVarId")}
    Modid "." CONID           -> QCONID  {cons("QConId")}
    Modid "." VARSYM          -> QVARSYM {cons("QVarSym")}
    Modid "." CONSYM          -> QCONSYM {cons("QConSym")}


module Haskell-Keywords
exports
  lexical syntax
    "case"      | "class"       | "data"        |
    "default"   | "deriving"    | "do"          |
    "else"      | "if"          | "import"      |
    "in"        | "infix"       | "infixl"      |
    "infixr"    | "instance"    | "let"         |
    "module"    | "newtype"     | "of"          |
```

```
    "then"      | "type"        | "where"       |
    "_"                                         -> ReservedId
    "as"        | "hiding"      | "qualified"   |
    "export"    | "label"       | "dynamic"     -> ReservedId0
    "forall"                                    -> ReservedId2

    ".." | ":" | "::" | "=" | "\\" | "|" | "<-" |
    "->" | "@" | "~" | "=>" |
    "-" | "!" | "." | "/\\" | "{" | "}" | "[" |
    "]" | "(" | ")" | "(#" | "#)" | ";" | "," | "`" -> ReservedOp


module Haskell-Identifier-Sorts
exports
  lexical syntax
    VARID               -> Varid
    ReservedId          -> Varid {reject}

    VARID               -> Tyvar
    ReservedId          -> Varid {reject}
    ReservedId2         -> Varid {reject}

  context-free syntax
    Vars "," Var        -> Vars         {cons("Snoc")}
    Qvar                -> Vars         {cons("Ins")}

  context-free syntax
    "(" ")"             -> Gcon         {cons("Unit")}
    "[" "]"             -> Gcon         {cons("EmptyList")}
    "(" ","+ ")"        -> Gcon         {cons("Product")}
    Qcon                -> Gcon

  context-free syntax

    %% variable identifiers

    Varid               -> Var          {cons("VarId")}
    Qvarid              -> Qvar
    Varid               -> Qvarid
    QVARID              -> Qvarid

    %% constructor identifiers

    Conid               -> Con          {cons("ConId")}
    Qconid              -> Qcon
    Conid               -> Qconid
    QCONID              -> Qconid
    CONID               -> Conid

  %% The following portion can be put into module Haskell-Infix
```

```
%% in order to factor out infix operators from the kernel language

context-free syntax

  %% infix operators

  Varop               -> Op           {cons("Op")}
  Conop               -> Op           {cons("ConOp")}

  %% variable operators

  Varsym              -> Varop
  Qvarsym             -> Qvarop
  Qvarsymm            -> Qvaropm
  Varsym              -> Qvarsym
  Qvarsym1            -> Qvarsym
  Varsymm             -> Qvarsymm
  Qvarsym1            -> Qvarsymm

  %% constructor operators

  Consym              -> Qconsym
  QCONSYM             -> Qconsym
  CONSYM              -> Consym
  Consym              -> Conop
  Qconsym             -> Qconop

  Qvarop                -> Qop
  Qconop                -> Qop
  Qvaropm               -> Qopm
  Qconop                -> Qopm

  %% make prefix symbols from infix symbols

  "(" Varsym ")"      -> Var          {cons("BinOp")}
  "(" Qvarsym ")"     -> Qvar         {cons("BinOp")}
  "(" Consym ")"      -> Con          {cons("BinCon")}
  "(" Qconsym ")"     -> Qcon         {cons("BinCon")}

  %% make infix symbols from prefix symbols

  "`" Varid "`"       -> Varop        {cons("PrefOp")}
  "`" Qvarid "`"      -> Qvarop       {cons("PrefOp")}
  "`" Qvarid "`"      -> Qvaropm      {cons("PrefOp")}
  "`" Conid "`"       -> Conop        {cons("PrefCon")}
  "`" Qconid "`"      -> Qconop       {cons("PrefCon")}

context-free syntax
  VARSYM              -> Varsym
  "-"                 -> Varsym       {cons("Subtraction")}
```

```
    "!"                 -> Varsym
    "."                 -> Varsym

    VARSYM              -> Varsymm
    "!"                 -> Varsymm
    "."                 -> Varsymm

    QVARSYM             -> Qvarsym1

  context-free syntax
    CONID               -> Modid
    CONID               -> Tycon
    Tycon               -> Qtycon
    QCONID              -> Qtycon
    Qtycon              -> Qtycls


module Haskell-Numbers
exports
  lexical syntax
    [0-9]                -> Digit
    [0-7]                -> Octit
    [0-9A-Fa-f]          -> Hexit

    Digit+               -> Decimal
    Octit+               -> Octal
    Hexit+               -> Hexadecimal

    Decimal              -> INTEGER
    [0] [Oo] Octal       -> INTEGER
    [0] [Xx] Hexadecimal -> INTEGER
  lexical restrictions
    INTEGER -/- [0-9]

  lexical syntax
    Decimal "." Decimal ([eE] [\-\+]? Decimal) -> RATIONAL
  lexical restrictions
    RATIONAL -/- [0-9]

  lexical syntax
    [] -> PRIMCHAR
    [] -> PRIMSTRING
    [] -> PRIMINTEGER
    [] -> PRIMFLOAT
    [] -> PRIMDOUBLE

    [] -> CLITLIT
    [] -> UNKNOWN


module Haskell-Strings
```

```
exports
  lexical syntax
    "'" CharChar "'"       -> CHAR
    "\"" StringChar* "\""  -> STRING

    ~[\0-\31\"\\]          -> StringChar
    Escape                 -> StringChar
    Gap                    -> StringChar

    [\\] [\ \t\n]+ [\\]    -> Gap

    [\\] (CharEsc | ASCII
         | Decimal
         | ([o] Octal)
         | [x] Hexadecimal) -> Escape

    [abfnrtv\\\"\'\&]       -> CharEsc

  lexical syntax
    "^" [A-Z\@\[\]\\\^\_]                   -> ASCII

    "NUL" | "SOH" | "STX" | "ETX" | "EOT" |
    "ENQ" | "ACK" | "BEL" | "BS"  | "HT"  |
    "LF"  | "VT"  | "FF"  | "CR"  | "SO"  |
    "SI"  | "DLE" | "DC1" | "DC2" | "DC3" |
    "DC4" | "NAK" | "SYN" | "ETB" | "CAN" |
    "EM"  | "SUB" | "ESC" | "FS"  | "GS"  |
    "RS"  | "US"  | "SP"  | "DEL"           -> ASCCI


module Haskell-Literals
exports
  context-free syntax
    INTEGER             -> Literal {cons("Int")}
    CHAR                -> Literal {cons("Char")}
    RATIONAL            -> Literal {cons("Float")}
    STRING              -> Literal {cons("String")}
    PRIMINTEGER         -> Literal {cons("PrimInt")}
    PRIMCHAR            -> Literal {cons("PrimChar")}
    PRIMSTRING          -> Literal {cons("PrimString")}
    PRIMFLOAT           -> Literal {cons("PrimFloat")}
    PRIMDOUBLE          -> Literal {cons("PrimDouble")}
    CLITLIT             -> Literal {cons("CLitLit")}
```

3.5 Modules

```
module Haskell-Modules
imports ExtraSepLists[Topdecl ";" Topdecls]
```

```
exports
  context-free syntax
    "module" Modid Exports?
            "where" Body        -> Module       {cons("Module")}
    Body                        -> Module       {cons("Program")}
    "{" Top "}"                 -> Body
    Topdecls                    -> Top          {cons("TopDecls")}
    Decl                        -> Topdecl
```

3.6 Declarations

```
module Haskell-Types
exports
  context-free syntax
    ("::" Type)?           -> OptSig

  context-free syntax
    Qtycon                 -> Gtycon
    "(" "->" ")"           -> Gtycon    {cons("TArrow")}

  context-free syntax
    {Type ","}+            -> Types
    Type "," {Type ","}+   -> Types2    {cons("Cons")}
    "forall" Tyvar* "."    -> Forall
    Type "=>"              -> Context

  context-free syntax
    Gtycon                 -> Type      {cons("TCon")}
    Tyvar                  -> Type      {cons("TVar")}
    Type Type              -> Type      {cons("TAppBin"),left}
    Type "->" Type         -> Type      {cons("TFunBin"),right}
    Forall Type            -> Type      {cons("Forall")}
    Forall Context Type    -> Type      {cons("ForallContext")}

    "(" Type ")"           -> Type      {bracket}

  context-free priorities
    Type Type              -> Type
  > Type "->" Type         -> Type
  > {Forall Type           -> Type
     Forall Context Type   -> Type}

  %% The following productions are syntactic sugar for
  %% [] Type and (,,,) Type ... Type

  context-free syntax
    "[" Type "]"           -> Type      {cons("TList")}
    "(" Types2 ")"         -> Type      {cons("TProd")}
```

```
    "(" ")"                -> Gtycon    {cons("TUnit")}
    "[" "]"                -> Gtycon    {cons("TList")}
    "(" ","+ ")"           -> Gtycon    {cons("TProduct")}


module Haskell-Type-Declarations
exports
  context-free syntax
    "type" Tycon Tyvar* "=" Type          -> Topdecl {cons("TypeDecl")}
    "data" Type "=" Constrs Deriving      -> Topdecl {cons("Data")}
    "newtype" Type "=" Newconstr Deriving -> Topdecl {cons("NewTypeDecl")}

  context-free syntax
    "deriving" Qtycls                   -> Deriving
    "deriving" "(" ")"                  -> Deriving
    "deriving" "(" {Qtycls ","}+ ")"    -> Deriving
                                        -> Deriving {cons("NoDeriving")}
  context-free syntax
    {Constr "|"}+                        -> Constrs

    Forall? Context? Conid Satype*       -> Constr {cons("ConstrDecl")}
    Forall? Context? Sbtype Conop Sbtype -> Constr {cons("InfixConstr")}

    Conid Type                           -> Newconstr
    Conid "{" Var "::" Type "}"          -> Newconstr

    Type                                 -> Satype
    "!" Type                             -> Satype
    Type                                 -> Sbtype
    "!" Type                             -> Sbtype

  context-free priorities
    Type      -> Satype
  > Type Type -> Type


module Haskell-Signature-Declarations
exports
  context-free syntax
    Signdecl       -> Decl
    Vars "::" Type -> Signdecl {cons("SignDecl")}
```

3.7 Expressions

```
module Haskell-Expressions
exports
  context-free syntax
    Exp                                  -> AnyExp
```

```
    Qvar                                -> Exp         {cons("Var")}
    Gcon                                -> Exp         {cons("Constr")}
    Literal                             -> Exp         {cons("Lit")}
    "_"                                 -> Exp         {cons("Wildcard")}
    "(" Exps2 ")"                       -> Exp         {cons("Product")}
    "(#" Exps "#)"                      -> Exp         {cons("Unboxed?")}
    "(" Exp ")"                         -> Exp         {bracket}

    {Exp ","}+                          -> Exps
    Exp "," {Exp ","}+                  -> Exps2       {cons("Cons")}
    Aexp+                               -> Fargs

  context-free priorities
    "~" Exp                              -> Exp        {cons("TILDE?")}
  > Qvar "@" Exp                         -> Exp        {cons("AT?")}
  > Exp                                  -> Aexp1
  > Exp "{" Fbinds "}"                   -> Exp        {cons("Labeled")}
  > Exp                                  -> Aexp
  > Exp Exp                              -> Exp        {cons("AppBin"),left}
  > Exp                                  -> Infixexp
  > Exp "::" Type                        -> Exp        {cons("Typed")}
  > {"\\" Fargs OptSig "->" Exp          -> Exp        {cons("Abs")}
     "let" Declbinds "in" Exp            -> Exp        {cons("Let")}
     "if" AnyExp "then" AnyExp "else" Exp -> Exp       {cons("If")}
     "case" AnyExp "of" Altslist         -> Exp        {cons("Case")}
    }
  > Exp                                  -> Exp10

%% Notes:
%% AnyExp is used to prevent priorities from forbidding expressions
%% where the do not cause ambiguities.
%% Fargs is used instead of Aexp+ because of a bug in the SDF2
%% normalizer; regular expression expansion does not take into
%% account symbols used only in priority rules.


module Haskell-Case-Alternatives
imports ExtraSepLists0[Alt ";" Alts]
exports
  context-free syntax
    "{" Alts "}"                       -> Altslist
    Infixexp OptSig "->" Exp           -> Alt   {cons("Alt")}
    Infixexp OptSig "->" Exp Where     -> Alt   {cons("AltW")}

    Infixexp OptSig Gdpat+   Where?    -> Alt   {cons("GdAlt")}
    "|" Quals "->" Exp                 -> Gdpat {cons("GdPat")}


module Haskell-Infix
exports
```

```
  context-free syntax
    "infix"               -> Infix      {cons("Infix")}
    "infixl"              -> Infix      {cons("InfixL")}
    "infixr"              -> Infix      {cons("InfixR")}
    INTEGER?              -> Prec
    {Op ","}+             -> Ops
    Infix Prec Ops        -> Fixdecl    {cons("FixDecl")}
    Fixdecl               -> Decl

    "(" Infixexp Qop ")"  -> Exp        {cons("LSection")}
    "(" Qopm Infixexp ")" -> Exp        {cons("RSection")}
  context-free priorities
    "-" Exp               -> Exp  {cons("Negation")}
  > "~" Exp               -> Exp
  , Exp                   -> Exp10
  > Exp Qop Exp           -> Exp  {cons("OpApp"),left}


module Haskell-Value-Definitions
imports ExtraSepLists0[Decl ";" Decls]
exports
  context-free syntax
    Valdef                 -> Decl
    Infixexp "=" Exp       -> Valdef    {cons("Valdef")}
    Infixexp "=" Exp Where -> Valdef    {cons("ValdefW")}

    Infixexp Gdrh+ Where?  -> Valdef    {cons("GdValdef")}
    "|" Quals "=" Exp      -> Gdrh      {cons("Guarded")}

    "where" Decllist       -> Where     {cons("Where")}
    Decllist               -> Declbinds
    "{" Decls "}"          -> Decllist
```

Chapter 4

Abstract Syntax

4.1 Summary

This chapter presents the abstract syntax of Haskell programs used in the transformations. The constructors of abstract syntax trees are declared by means of algebraic signatures. Module `AHaskell` gives a summary of the constructors most used in the transformations.

```
module AHaskell
signature
  sorts Decl Constr Type Exp Alt
  (* subsorts TVar < Type *)
  constructors
    Program    : List(Decl)                          -> Program

    Data       : Type * List(Constr) * Deriving      -> Decl
    ConstrDecl : Option(Forall) * Option(Context)
                 * String * List(Type)               -> Constr

    SignDecl   : Vars * Type                         -> Decl
    Valdef     : Exp * Exp                           -> Decl

    TVar       : String                              -> TVar
    TCon       : String                              -> Type
    TApp       : Type * List(Type)                   -> Type
    TFun       : List(Type) * Type                   -> Type
    Forall     : List(String) * Type                 -> Type

    Typed      : Exp * Type                          -> Exp
    Var        : String                              -> Exp
    Constr     : String                              -> Exp
    Lit        : Literal                             -> Exp
    Abs        : List(Exp) * Option(Type) * Exp      -> Exp
    App        : Exp * List(Exp)                     -> Exp
    Let        : List(Decl) * Exp                    -> Exp
    Case       : Exp * List(Alt)                     -> Exp
```

```
    Alt        : Exp * Option(Type) * Exp              -> Alt
    TAbs       : List(TVar) * Exp                      -> Exp
    TInst      : Exp * List(Type)                      -> Exp

    Build      : Type * Exp                            -> Exp
    Cata       : Type * Type * List(Exp)               -> Exp
```

4.2 Haskell Signature

4.2.1 Haskell-Kernel

```
module Haskell-Kernel
imports Haskell-Identifiers
        Haskell-Identifier-Sorts
        Haskell-Literals
        Haskell-Modules
        Haskell-Types
        Haskell-Type-Declarations
        Haskell-Signature-Declarations
        Haskell-Expressions
        Haskell-Case-Alternatives
        Haskell-Value-Definitions
        Haskell-Infix
        Haskell-Build-Cata
```

4.3 Literals

4.3.1 Haskell-Identifier-Sorts

```
module Haskell-Identifier-Sorts
signature
  constructors
    BinOp  : a -> a
    PrefOp : a -> a
```

4.3.2 Haskell-Identifiers

```
module Haskell-Identifiers
signature
  constructors
    QVarId  : Modid * VARID           -> QVARID
    QConId  : Modid * CONID           -> QCONID
    QVarSym : Modid * VARSYM          -> QVARSYM
    QConSym : Modid * CONSYM          -> QCONSYM
```

4.3.3 Haskell-Literals

```
module Haskell-Literals
signature
  constructors
    Int         : String -> Literal
    Char        : String -> Literal
    Float       : String -> Literal
    String      : String -> Literal
    PrimInt     : String -> Literal
    PrimChar    : String -> Literal
    PrimString  : String -> Literal
    PrimFloat   : String -> Literal
    PrimDouble  : String -> Literal
    CLitLit     : String -> Literal
```

4.4 Modules

4.4.1 Haskell-Modules

```
module Haskell-Modules
signature
  constructors
    Program  : List(Decl)                       -> Program
    Module   : Modid * Opt(Export) * List(Decl) -> Module
    Body     : List(Decl)                       -> Module
    TopDecls : List(Decl)                       -> Module
```

4.5 Declarations

4.5.1 Haskell-Types

```
module Haskell-Types
signature
  sorts TVar Type
  (* subsorts TVar < Type *)
  constructors
    TArrow        : Gtycon
    TUnit         : Gtycon
    TList         : Gtycon
    TProduct      : Gtycon

    TVar          : String                  -> TVar
    TCon          : String                  -> Type
    TApp          : Type * List(Type)       -> Type
    TFun          : List(Type) * Type       -> Type
    Forall        : List(TVar) * Type       -> Type

    ForallContext : List(TVar) * Context * Type -> Type
```

```
    TList         : Type                    -> Type
    TProd         : List(Type)              -> Type
    TAppBin       : Type * Type             -> Type
    TFunBin       : Type * Type             -> Type
```

4.5.2 Haskell-Type-Declarations

```
module Haskell-Type-Declarations
signature
  constructors
    Data        : Type * List(Constr) * Deriving   -> Decl

    ConstrDecl  : Option(Forall) * Option(Context)
                  * Conid * List(Type)             -> Constr

    TypeDecl    : Tycon * List(Tyvar) * Type    -> Decl
    NewTypeDecl : Type * Newconstr * Deriving   -> Decl

    NoDeriving  : Deriving

    InfixConstr : Option(Forall) * Option(Context)
                  * Type * Conop * Type -> Constr
```

4.5.3 Haskell-Signature-Declarations

```
module Haskell-Signature-Declarations
signature
  constructors
    SignDecl : Vars * Type -> Topdecl
```

4.5.4 Haskell-Value-Definitions

```
module Haskell-Value-Definitions
signature
  constructors
    Valdef   : Exp * Exp                -> Decl
    ValdefW  : Exp * Exp * Where        -> Decl
    GdValdef : Exp * List(Gdrh) * Where -> Decl
    Guarded  : Quals * Exp              -> Gdrh
    Where    : List(Decl)               -> Where
```

4.6 Expressions

4.6.1 Haskell-Expressions

```
module Haskell-Expressions
signature
  constructors
    Typed  : Exp * Type                     -> Exp
```

```
    Var    : String                          -> Exp
    Constr : String                          -> Exp
    Lit    : Literal                         -> Exp

    Abs    : List(Exp) * Option(Type) * Exp -> Exp
    App    : Exp * List(Exp)                 -> Exp
    Let    : List(Decl) * Exp                -> Exp
    Case   : Exp * List(Alt)                 -> Exp

    TAbs   : List(TVar) * Exp                -> Exp
    TInst  : Exp * List(Type)                -> Exp

    (* sugar *)

    Product: List(Exp)                       -> Exp
    AppBin : Exp * Exp                       -> Exp
    If     : Exp * Exp * Exp                 -> Exp

    Wildcard:                                   Exp
    Unboxed: List(Exp)                       -> Exp
    TILDE  : Exp                             -> Exp
    AT     : Qvar * Exp                      -> Exp
    Labeled: Exp * Fbinds                    -> Exp
```

4.6.2 Haskell-Case-Alternatives

```
module Haskell-Case-Alternatives
signature
  constructors
    Alt   : Exp * Option(Type) * Exp       -> Alt

    AltW  : Exp * Option(Type) * Exp * Where                 -> Alt
    GdAlt : Exp * Option(Type) * List(Gdpat) * Option(Where) -> Alt
    GdPat : Quals * Exp                                      -> Gdpat
```

4.6.3 Haskell-Infix

```
module Haskell-Infix
signature
  constructors
    Infix    : Infix
    InfixL   : Infix
    InfixR   : Infix
    FixDecl  : Infix * Prec * Ops -> Fixdecl

    LSection : Infixexp * Qop     -> Exp
    RSection : Qopm * Infixexp    -> Exp
    Negation : Exp                -> Exp
    OpApp    : Exp * Qop * Exp    -> Exp
```

4.6.4 Haskell-Build-Cata

```
module Haskell-Build-Cata
imports Haskell-Kernel
signature
  constructors
    Build : Type * Exp                          -> Exp
    Cata  : Type * Type * List(Exp)             -> Exp
```

Chapter 5

Pretty-Printing

5.1 Pretty-Printing Haskell

This chapter presents the specification of a pretty-printer for the abstract syntax of Haskell. The pretty-printer is a mapping from the abstract syntax trees to Box expressions, a language independent format for pretty-printing [15]. Box expressions can be formatted for a variety of presentation targets including text, HTML and LaTeX.

5.1.1 PP-Haskell-Kernel

```
module PP-Haskell-Kernel
imports lib abox-ext
        PP-Haskell-Identifiers
        PP-Haskell-Identifier-Sorts
        PP-Haskell-Literals
        PP-Haskell-Modules
        PP-Haskell-Types
        PP-Haskell-Type-Declarations
        PP-Haskell-Signature-Declarations
        PP-Haskell-Expressions
        PP-Haskell-Case-Alternatives
        PP-Haskell-Value-Definitions
        PP-Haskell-Infix
        PP-Haskell-Qualifiers
strategies

  main = iowrap(pp-haskell)

  pp-haskell = topdown(repeat(PP-HSe <+ PP-HS))

rules

  PP-HS : None -> EmptyBox()
  PP-HS : Some(x) -> x
```

5.2 Literals

5.2.1 PP-Haskell-Identifier-Sorts

```
module PP-Haskell-Identifier-Sorts
imports Haskell-Identifier-Sorts
rules

PP-HS : BinOp(x)  -> Parens(S(x))

PP-HS : PrefOp(x) -> H0([S("`"), S(x), S("`")])
```

5.2.2 PP-Haskell-Identifiers

```
module PP-Haskell-Identifiers
imports abox Haskell-Identifiers
rules

PP-HS : QVarId(m, v)  -> H([SOpt(HS,0)],[m, S("."), v])
PP-HS : QConId(m, v)  -> H([SOpt(HS,0)],[m, S("."), v])
PP-HS : QVarSym(m, v) -> H([SOpt(HS,0)],[m, S("."), v])
PP-HS : QConSym(m, v) -> H([SOpt(HS,0)],[m, S("."), v])
```

5.2.3 PP-Haskell-Literals

```
module PP-Haskell-Literals
imports Haskell-Literals
rules

  PP-HS : Int(x)        -> S(x)
  PP-HS : Char(x)       -> S(x)
  PP-HS : Float(x)      -> S(x)
  PP-HS : String(x)     -> S(x)
  PP-HS : PrimInt(x)    -> S(x)
  PP-HS : PrimChar(x)   -> S(x)
  PP-HS : PrimString(x) -> S(x)
  PP-HS : PrimFloat(x)  -> S(x)
  PP-HS : PrimDouble(x) -> S(x)
  PP-HS : CLitLit(x)    -> S(x)
```

5.3 Modules

5.3.1 PP-Haskell-Modules

```
module PP-Haskell-Modules
imports Haskell-Modules
rules

PP-HS : Module(x, None, body) ->
        V1([H1([S("module"), S(x), S("where"), S("{")]),
```

```
            body,
            S("}")])

PP-HS : Program(body) ->
        V1([S("{"), body, S("}")])

PP-HS : Body(ds)     -> V1(<hpost-sep-list(id,!S(";"))> ds)
PP-HS : TopDecls(ds) -> V1(<hpost-sep-list(id,!S(";"))> ds)
```

5.4 Declarations

5.4.1 PP-Haskell-Signature-Declarations

```
module PP-Haskell-Signature-Declarations
imports Haskell-Signature-Declarations
rules

PP-HS : SignDecl(xs, tp) ->
        H1([H1(<sep-list(PP-HS <+ MkS,!",")> xs), S("::"), tp])
```

5.4.2 PP-Haskell-Type-Declarations

```
module PP-Haskell-Type-Declarations
imports Haskell-Type-Declarations abox
rules

PP-HS : Type(tycon, tyvars, t) ->
        H1([Keyword(S("type")), tycon, H1(tyvars), S("="), t])

PP-HS : Data(t, cs, der) ->
        H1([Keyword(S("data")), t,
            V0([V0(<prebars(!S("="))> cs), der])])

PP-HS : NewType(tp, nconstr, der) ->
        H1([Keyword(S("newtype")), tp, S("="), nconstr, der])

PP-HS : NoDeriving() -> EmptyBox()

PP-HS : ConstrDecl(forall, context, conid, tps) ->
        H1([forall, context, S(conid), H1(<map(MkParens)>tps)])

PP-HS : InfixConstr(forall, context, tp, conop, tp) ->
        H1([forall, context, tp, S(conop), tp])
```

5.4.3 PP-Haskell-Types

```
module PP-Haskell-Types
imports Haskell-Types abox
rules
```

```
PP-HS : TArrow -> S("(->)")

PP-HS : TUnit -> S("()")

PP-HS : TList -> S("[]")

PP-HS : TProduct -> S("(*** product type ***)")
        // Note: depends on context

PP-HS : TCon(x) -> S(x)
PP-HS : TVar(x) -> S(x)
PP-HS : TAppBin(x, y) -> H1([x, y])
PP-HS : TFunBin(x, y) -> H1([Parens(x), S("->"), y])

PP-HS : TApp(t, ts) ->
        H1([t, H1(ts)])

PP-HS : TFun(ts, t) ->
        H1([H1(<sep-list(MkParens,!"->")> ts), S("->"), t])


PP-HS : Forall(as, t) ->
        H1([S("forall"), H1(<commas> as), S("."), t])

PP-HS : ForallContext(as, t, t') ->
        H1([S("forall"), H1(<commas> as), S("."), t, S("=>"), t'])

PP-HS : TList(t) -> H1([S("["),t,S("]")])

PP-HS : TProd(ts) -> Parens(H1(<commas>ts))
```

5.4.4 PP-Haskell-Value-Definitions

```
module PP-Haskell-Value-Definitions
imports Haskell-Value-Definitions
rules
PP-HS : Valdef(e1, e2) ->
        H1([e1, S("="), e2])

PP-HS : ValdefW(e1, e2, wr) ->
        H1([e1, S("="), V0([e2, wr])])

PP-HS : GdValdef(e, gs, wr) ->
        H1([e, V0([gs, wr])])

PP-HS : Guarded(qs, e) ->
        H1([S("|"), H1(<sep-list(id,!",")>qs), e])

PP-HS : Where([]) ->
        EmptyBox()
```

```
PP-HS : Where(ds) ->
        H1([S("where"), S("{"), V0(<semicolons> ds), S("}")])
        where <not([])> ds
```

5.5 Expressions

5.5.1 PP-Haskell-Case-Alternatives

```
module PP-Haskell-Case-Alternatives
imports Haskell-Case-Alternatives
rules
PP-HS : Alt(e, sig, e2) ->
        ALT(H1([H1([e, <PP-SIG> sig]), H1([S("->"), e2])]),
            V0([H1([e, <PP-SIG> sig]), H1([S("->"), e2])]))

PP-HS : AltW(e, sig, e2, wr) ->
        V0([H1([e, <PP-SIG> sig]), V0([H1([S("->"), e2]), wr])])

PP-HS : GdAlt(e, sig, pats, wr) ->
        V0([H1([e, <PP-SIG> sig, V0(pats)]), wr])

PP-HS : GdPat(quals, e) ->
        H1([S("|"), <sep-list(id,!",")> quals, S("->"), e])

PP-SIG : None -> None
PP-SIG : Some(sig) -> H1([S("::"), sig])
```

5.5.2 PP-Haskell-Expressions

```
module PP-Haskell-Expressions
imports Haskell-Expressions Haskell-Build-Cata

rules

PP-HS : Var(x) -> S(x)
        where <is-string> x

PP-HS : Var(x) -> x
        where <not(is-string)> x

PP-HS : Constr(x) -> S(x)
        where <is-string> x

PP-HS : Constr(x) -> x
        where <not(is-string)> x

PP-HS : Lit(x) -> S(x)
        where <is-string> x
```

```
PP-HS : Lit(x) -> x
        where <not(is-string)> x

PP-HS : Wildcard -> S("_")

PP-HS : Product(es) ->
        H0([S("("), H1(<commas> es), S(")")])

PP-HS : Unboxed(es) ->
        H0([S("(#"), H1(<commas> es), S("#)")])

PP-HS : TILDE(e) -> H0([S("~"), e])

PP-HS : AT(x, e) -> H0([x, S("@"), e])

PP-HS : Labeled(e, fbnds) ->
        H0([e, S("{"), H1(<commas> fbnds), S("}")])

PP-HS: AppBin(e1, e2) ->
        H0([e1, Parens(e2)])

PP-HSe: App(Var(BinOp(op)), [e1, e2]) ->
        ALT(Parens(H1([e1, S(op), e2])),
            ALT(H0([Var(BinOp(op)), <map(MkParens)>[e1, e2]]),
                V0([Var(BinOp(op)),
                    Indent(V0(<map(MkParens)>[e1, e2]))]))))

PP-HS : App(e, es) ->
        ALT(H0([e, <map(MkParens)>es]),
            V0([e, Indent(V0(<map(MkParens)>es))]))

PP-HS : Typed(e, t) ->
        Parens(H1([e, S("::"), t]))

PP-HS : Abs(args, sig, e) ->
        Parens(V0([H1([H0([S("\\\\"), H1(args)]),
                       <PP-SIG> sig, S("->")]),
                   e]))

PP-HS : TAbs(ts, e) ->
        V0([H1([H0([S("/\\\\"), H1(ts)]), S("->")]), e])

PP-HS : Let(decls, e) ->
        V0([H1([S("let"), S("{"),
                V0(<semicolons> decls),
                S("}")]),
            H1([S(" in"), e])])

PP-HS : If(e1, e2, e3) ->
```

```
        V0([H1([S("if"), e1, S("then")]),
            Indent(e2), S("else"), Indent(e3)])

PP-HS : Case(e, alts) ->
        V0([H1([S("case"), e, S("of")]),
            V0(<presemicolons(!S("{"))> alts),
            S("}")])

PP-HS : TInst(e, t) ->
        H1([e, H0([S("["), t, S("]")])])

PP-HS : Build(t, e) ->
        H0([S("build"), Parens(V0([H0([t, S(",")]), e]))])

PP-HS : Cata(t1, t2, es) ->
        ALT(H0([S("cata"), S("["), t1, S("]"), S("["), t2, S("]"),
                Parens(H1(<post-commas> es))]),
            V0([H0([S("cata"), S("["), t1, S("]"), S("["), t2, S("]")]),
                Indent(Parens(V0(<post-commas> es)))]))
```

5.5.3 PP-Haskell-Infix

```
module PP-Haskell-Infix
imports Haskell-Infix
rules

PP-HS : Infix  -> S("infix")
PP-HS : InfixL -> S("infixl")
PP-HS : InfixR -> S("infixr")

PP-HS : FixDecl(i, p, ops) ->
        H1([i, p, H1(<sep-list(id,!",")> ops)])

PP-HS : LSection(e, op) ->
        Parens(H1([e, S(op)]))

PP-HS : RSection(op, e) ->
        Parens(H1([S(op), e]))

PP-HS : Negation(e) ->
        H0([S("-"), e])

PP-HS : OpApp(e1, op, e2) ->
        Parens(H1([e1, S(op), e2]))
```

Chapter 6

Intermediate Formats

6.1 HS-Check

The transformation components defined in this paper all work on Haskell programs in abstract syntax form. However, most components accept only a subset of the entire language. In this module these subsets are characterized by means of recursive patterns [32]. These strategies can be used to check conformance of intermediate results to one of the subsets.

```
module HS-Check
imports Haskell-Kernel lib
```

The patterns are extended to report violations of the format. The constructor `Error` tags a subterm as erroneous, with a string indicating the type of the error. The rule `MkError` tags a term with this constructor and also prints a message to stderr.

```
signature
  constructors
    Error : String * a -> a
rules
  MkError(s) : x -> Error(<s>(), x) where <debug(s)> x
```

6.1.1 Fully Typed Programs

Type expressions consist of type variables, type constructors (functors), function types (n-ary), type quantification, and type application. A type constructor (`TyCon`) is used in data type declarations.

```
strategies

  type(t) =
    TVar(id) +
    TCon(id) +
    TFun(list(t), t) +
    Forall(list(TVar(id)), t) +
    TApp(t, list(t))
```

```
TyCon =
  TCon(id) + TApp(TCon(id), list(TVar(id)))

Type = rec t(type(t) <+ MkError(!"Not a type:"))
```

A fully typed expression is a typed atom, an abstraction over variables (not patterns), a case expression with alternatives ranging over simple patterns (not nested), let bindings or n-ary application, type abstraction, or type instantiation.

```
AExp = Var(id) + Constr(id) + Lit(id)

atom(t) = Typed(AExp, t)

TypedVar = Typed(Var(id),Type)

TypedAtom = atom(Type)

simple-pattern(var) =
  Constr(id) +
  App(Constr(id), list(var))

TypedPat =
  simple-pattern(TypedVar)
  <+ MkError(!"Not a TypedPat: ")

alt(e, t, pat) =
  Alt(pat, option(t), e)
  <+ MkError(!"Not an alt: ")

exp(e, t, pat, var) =
  Abs(list(var), option(t), e) +
  Case(e, list(alt(e, t, pat))) +
  Let(list(decl(e, t)), e) +
  App(e, list(e)) +
  TAbs(list(TVar(id)), e) +
  TInst(e, list(t))

TypedExp =
  rec e((TypedAtom + exp(e, Type, TypedPat, TypedVar))
        <+ MkError(!"Not a TypedExp: "))
```

A program consists of a list of top-declarations, which are data type definitions, signature declarations or value definitions.

```
decl(e, t) =
  Valdef(Var(id), e) +
  SignDecl(list(id), t)

topdecl(e, t) =
```

```
    decl(e, t) +
    Data(TyCon, list(ConstrDecl(None,None,id,list(t))), id)

  topdecls(td) =
    TopDecls(list(td <+ MkError(!"Not a topdecl: ")))

  hs-program(tds) =
    Module(id, id, tds) +
    Program(tds)

  hs-typed = hs-program(topdecls(topdecl(TypedExp, Type)))
```

6.1.2 Partially Typed Programs

Input programs do not have to be fully typed, i.e., atomic expressions (variables, constructors and literals) and variable declarations in abstractions and case alternatives can occur without type annotation. In addition, input programs can contain the binary versions of some n-ary constructors (`AppBin`, `TFunBin`, `TAppBin`), can contain infix operator applications (`OpApp`), and some syntactic sugar (`If`).

```
  pre-type(t) =
    TFunBin(t, t) +
    TAppBin(t, t)

  PreType =
    rec t((type(t) + pre-type(t))
          <+ MkError(!"Not a PreType: "))

  PreVar =
    Var(id) +
    Typed(Var(id), PreType)

  PrePat =
    simple-pattern(PreVar)
    + rec x(AppBin(x, PreVar) + Constr(id))
    <+ MkError(!"Not a PrePat: ")

  pre-exp(e) =
    OpApp(e, id, e) +
    AppBin(e, e) +
    Negation(e) +
    If(e, e, e)

  PreExp =
    rec e((AExp + atom(PreType) + pre-exp(e) +
           exp(e, PreType, PrePat, PreVar))
          <+ MkError(!"Not a PreExp:"))

  PreTycon =
```

```
    TyCon +
    rec x(TAppBin(x, TVar(id)) + TCon(id))

  pre-topdecl(e, t) =
    Data(PreTycon, list(ConstrDecl(None,None,id,list(t))), id) <+
    topdecl(e, t)

  hs-input =
    hs-program(topdecls(pre-topdecl(PreExp, PreType)))
```

6.1.3 Output Language

The output of the warm fusion transformation is again a fully typed program, but can in addition contain applications of `Build` and `Cata`.

```
  ext-exp(e, t) =
    Cata(t, t, list(e)) +
    Build(t, e)

  ExtExp =
    rec e((TypedAtom + exp(e, Type, TypedPat, TypedVar) +
           ext-exp(e, Type))
          <+ MkError(!"Not an ExtExp: "))

  hs-output =
    hs-program(topdecls(pre-topdecl(ExtExp, Type)))
```

6.1.4 Components

The format checkers for the three subsets are defined as follows:

```
  hs-input-component = iowrap(hs-input)

  hs-typed-component = iowrap(hs-typed)

  hs-output-component = iowrap(hs-output)
```

Chapter 7

Basic Transformation Utilities

7.1 Haskell-Lib

The Haskell-Lib is a collection of utilities for transforming Haskell programs not specific for warm fusion. Module `Haskell-Variables` defines strategies for manipulating variables in expressions (bound variable renaming, free variable extraction, substitution, unification). Module `Haskell-Type-Projection` defines strategies for deriving types from fully typed expressions and for stripping type annotations from typed programs. Module `Haskell-Data-Definitions` defines strategies for storing data type declarations in and retrieving them from a symbol table.

```
module Haskell-Lib
imports Haskell-Variables
        Haskell-Type-Projection
        Haskell-Data-Definitions
```

7.2 Haskell-Variables

This module defines strategies for manipulating expressions with (bound) variables by instantiating several generic strategies from the Stratego library. (See [34]for an introduction into these strategies). The `*vars` strategies extract the free variables from an expression. The `*rename` strategies rename all bound variables in an expression to new unique names. The `*subst` strategies substitute expressions for variables in an expression given a list of pairs of variables and expressions. The strategy `tpunify` tries to unify two type expression, producing a substitution if successful. The generic strategies are parameterized with strategies identifying sub-terms representing variables and binding constructs. These parameter strategies are defined using rules.

```
module Haskell-Variables
imports Haskell-Kernel lib substitution unification
rules
```

```
  IsVar(s)  : Var(x) -> Var(<s> Var(x))
  ExpVar    : Typed(Var(x),_) -> Var(x)
  ExpVar    : Var(x) -> Var(x)
  ExpVars   : Var(x) -> [Var(x)]

  ExpBnd    : Abs(xs, _, _) -> <map(ExpVar)> xs
  ExpBnd    : Alt(App(c, xs), t, e) -> <map(ExpVar)> xs
  ExpBnd    : Let(decls, e) -> <filter(DeclVar)> decls

  DeclVar   : Valdef(Var(x), e) -> Var(x)

  IsTVar(s) : TVar(x) -> TVar(<s> TVar(x))
  TpVar     : TVar(x) -> TVar(x)
  TpVars    : TVar(x) -> [TVar(x)]

  TpBnd     : Forall(as, t) -> as
  TpBnd     : TAbs(as, e) -> as

strategies

  VarName = ExpVar; \ Var(x) -> x \

  expvars = free-vars(ExpVars, ExpBnd)
  tpvars  = free-vars(TpVars, TpBnd)

  exprename = rename(IsVar, ExpBnd)
  tprename  = rename(IsTVar, TpBnd)
  etrename = exprename; tprename

  expsubst = substitute(Typed(Var(id),id) + Var(id), etrename)
  tpsubst  = substitute(TVar(id), tprename)
  tpsubst'' = substitute(TVar(id))
  expsubst'(lst) = split(lst, id); expsubst
  tpsubst'(lst) = split(lst, id); tpsubst

  tpunify = unify(TVar(id))

  hs-rename-component = iowrap(etrename)
```

7.3 Haskell-Type-Projection

```
module Haskell-Type-Projection
imports Haskell-Kernel Haskell-Variables
        Haskell-Normalize WF-Rules lib
```

7.3.1 Type Extraction

The type strategy maps a fully typed expression to its type. Since only atoms (variables, constructors and literals) are annoted with types, a little type manipulation is needed to compute the right type.

```
strategies

  type = rec x(GetType(x))

rules

  GetType(s) : Typed(x, t) -> t

  GetType(s) : Abs(xs, t, e) -> TFun(<map(s)> xs, <s> e)

  GetType(s) : App(e, es) -> <try(Uncurry)> TFun(ts1, t0)
               where <s> e => TFun(ts0, t0);
                     <zip-tail>(<map(s)>es, ts0) => ts1

  GetType(s) : Case(e1, [Alt(e2, t, e3) | as]) -> <s> e3

  GetType(s) : Let(decls, e) -> <s> e

  GetType(s) : TAbs(as, e) -> Forall(as, <s> e)
  GetType(s) : TInst(e, ts) -> <try(TBeta)> TApp(<s> e, ts)

  GetType(s) : Cata(t1, t2, es) -> TFun([t1], t2)
  GetType(s) : Build(t, e) -> t
```

7.3.2 Type Stripping

Fully typed programs can be turned into untyped programs by stripping off types from all atoms and variable declarations in abstractions. Also the type results in abstractions and case alternatives should be thrown away. The following transformation achieves this.

```
strategies

  strip-types = bottomup(try(StripT1 + StripT2 + StripT3))

  strip-types-component = iowrap(strip-types)

rules

  StripT1 : Typed(x, t) -> x

  StripT2 : Abs(xs, t, e) -> Abs(xs, None, e)

  StripT3 : Alt(e1, t, e2) -> Alt(e1, None, e2)
```

7.3.3 Type Manipulation

The domain of a function is its first argument (`dom`) and the range of a function is the rest of its argument and the result type (`range`).

```
rules

  dom    : TFun([t1 | ts], t3) -> t1
  range  : TFun([t1], t2) -> t2
  range  : TFun([t1, t2 | ts], t3) -> TFun([t2 | ts], t3)
```

A type is generalized by quantifying over all its free type variables. A polymorphic type is instantiated by renaming it (to create new unique variable names) and then leaving off its outer quantifier.

```
rules

  Generalize : t -> Forall(as, t) where <tpvars> t => as

strategies

  instantiate = tprename; try( \ Forall(_,t) -> t \ )
```

7.4 Haskell-Data-Definitions

Several of the transformations in the warm fusion algorithm generate expressions based on type information. In order to access the data type definitions at arbitrary places, these are stored in a symbol table.

```
module Haskell-Data-Definitions
imports Haskell-Kernel
```

7.4.1 Storing Data Type Definitions

The strategy `collect-data-defs` stores each data definition in the program in the `tycon` table. The table maps the name of the data type to the pair of formal type parameters and the constructor declarations.

```
strategies

  collect-data-defs =
    where(<create-table> "tycon");
    map(try(StoreDataDef));
    where(<table-keys> "tycon")

  StoreDataDef =
    ?Data(TCon(x), cs, _);
    <table-put> ("tycon", x, ([], cs))

  StoreDataDef =
    ?Data(TApp(TCon(x), as), cs, _);
    <table-put> ("tycon", x, (as, cs))
```

7.4.2 Retrieving Constructor Declarations

Given a type constructor (possibly applied to a list of actual type parameters) the strategy `get-constructors` produces a list of constructor declarations (instantiated to the actual parameters).

```
rules

  get-constructors :
    TCon(c) -> cs
    where <table-get>("tycon", c) => ([], cs)

  get-constructors :
    TApp(TCon(c), ts) -> <tpsubst> (as, ts, cs)
    where <table-get> ("tycon", c) => (as, cs)
```

Given the name of a constructor and its data type produce the list of argument types of the constructor.

```
strategies

  get-constructor-arg-types =
    (id, get-constructors); get-constructor

rules

  get-constructor :
    (c, arms) -> ts
    where <fetch(ConstrDecl(id,id,?c,?ts))> arms
```

Chapter 8

Normalization

8.1 Haskell-Normalize

The syntax definition of Haskell defines many operations as binary (curried) operations. For the purpose of transformation an n-ary representation is more convenient. In this chapter a normalization of binary to n-ary representation is specified. This normalization is achieved in two phases. In the first phase binary operations are mapped to their n-ary counterparts. In the second phase, curried applications of such n-ary applications are uncurried, i.e., the argument lists are collapsed.

```
module Haskell-Normalize
imports lib Haskell-Kernel
strategies

  normalize =
    topdown(try(SubtractionHack <+ U2N + B2N + If2Case));
    uncurry

rules

  B2N : AppBin(e1, e2) -> App(e1, [e2])

  B2N : TAppBin(e1, e2) -> TApp(e1, [e2])

  B2N : TFunBin(t1, t2) -> TFun([t1], t2)

  B2N : OpApp(e1, op, e2) -> App(Var(BinOp(op)), [e1, e2])

  SubtractionHack :
        AppBin(e1, Negation(e2)) -> App(Var(BinOp("-")), [e1, e2])

  U2N : Negation(e) -> App(Var("-"), [e])

  If2Case :
        If(e1, e2, e3) ->
```

```
        Case(e1,[Alt(Constr("True"),None,e2),
                 Alt(Constr("False"),None,e3)])
```

Normalizing nested applications of n-ary constructors

```
strategies

  uncurry   = topdown(repeat(Uncurry))

rules

  Uncurry : TFun([], t) -> t

  Uncurry : TApp(t, []) -> t

  Uncurry : App(e, []) -> e

  Uncurry : Abs([], t, e) -> e

  Uncurry : Abs(xs, t1, Abs(ys, t2, e)) ->
            Abs(<conc> (xs, ys), t1, e)

  Uncurry : TFun(ts1, TFun(ts2, t)) -> TFun(<conc>(ts1, ts2), t)

  Uncurry : App(App(f, args1), args2) -> App(f, <conc>(args1, args2))

  Uncurry : TApp(TApp(f, args1), args2) -> TApp(f, <conc>(args1, args2))

  Uncurry': TFun(ts, t) -> TApp(TArrow, <conc>(ts, [t]))

  Uncurry : TApp(TArrow, ts) -> TFun(<init> ts, <last> ts)

strategies

  hs-normalize-component = iowrap(normalize)
```

Chapter 9

Typechecking

9.1 Haskell-Typecheck

For the purpose of the warm fusion transformation, programs are required to be fully typed, i.e., for every subexpression it should be possible to infer its type without reference to declarations in the context. Since writing fully typed programs is untractable for humans, a typechecker is defined that turns a partially typed program (in the `hs-input` format) into a fully typed program (in the `hs-typed` format).

```
module Haskell-Typecheck
imports Haskell-Kernel Haskell-Lib lib

strategies

  main = iowrap(tc-module)

  tc-module = Module(id, id, TopDecls(typecheck))

  tc-module = Program(TopDecls(typecheck))

  typecheck =
    where(collect-data-defs);
    where(collect-signatures => env);
    map(try(annotate-def(!env)))
```

Top-level signatures

```
strategies

  collect-signatures =
    filter(get-signature);
    concat

rules

  get-signature :
```

```
    SignDecl(fs, t) -> <map(\ f -> (f, t')\ )> fs
    where <Generalize> t => t'

  get-signature :
    Data(t, cs, _) ->
    <map({c, ts: ?ConstrDecl(_,_,c,ts);
                 !(c, <(<Nil>ts;!t <+ !TFun(ts, t)); Generalize> ())})> cs
```

Distributing types over bodies of value definitions

```
signature
  constructors
    Tenv : Exp * List(Prod([String, Type])) -> Exp

strategies

tc-exp = rec x(tc(x) <+ debug; RmTenv <+ debug)

rules

RmTenv : Tenv(e, env) -> e

annotate-def(env) :
        Valdef(Var(f), e) ->
        Valdef(Var(f), <tc-exp> Tenv(Typed(e, t), <env>()))
        where <lookup; instantiate>(f, <env>()) => t

tc(s) : Tenv(Typed(Typed(e, t), None), env) -> Tenv(Typed(e, t), env)

tc(s) : Tenv(Typed(Var(x), t0), env) ->
        <tpsubst> (sbs, Typed(Var(x), t1))
        where <lookup; instantiate>(x, env) => t1;
              <[(id,None)]; ![] <+ tpunify> [(t1, t0)] => sbs

tc(s) : Tenv(Typed(Constr(x), t0), env) ->
        <tpsubst> (sbs, Typed(Constr(x), t1))
        where <lookup; instantiate>(x, env) => t1;
              <[(id,None)]; ![] <+ tpunify> [(t1, t0)] => sbs

tc(s) : Tenv(Typed(Lit(Int(x)), _), env) ->
        Typed(Lit(Int(x)), TCon("Int"))

tc(s) : Tenv(Typed(App(e, es), t), env) ->
        <tpsubst> (sbs, App(e', es'))
        where
          <s> Tenv(Typed(e, None), env) => e';
          <type> e' => TFun(ts, t');
          <map(\x -> <s>Tenv(Typed(x,None),env)\ )> es => es';
          <map(type)> es' => ts';
          <zipr(id); (tpunify <+ debug(!"types do not match: "); FAIL)>
```

```
            (ts, ts') => sbs

tc(s) : Tenv(Typed(Abs(xs, t, e), TFun(ts, t')), env) ->
        Abs(ys, Some(t'), <s> Tenv(Typed(e, t'), <conc>(env', env)))
        where <zip(\ (x,t) -> Typed(x,t)\ )>(xs, ts) => ys;
              <map(split(VarName, type))> ys => env'

tc(s) : Tenv(Typed(Case(e, alts), t), env) ->
        Case(e', alts')
        where <s> Tenv(Typed(e, None), env) => e';
              <type> e' => t';
              <map(\alt -> <s> Tenv(Typed(alt, TFun([t'], t)), env) \ )>
                alts => alts'

tc(s) : Tenv(Typed(Alt(Constr(c), t0, e), TFun([t1], t2)), env) ->
        Alt(Constr(c), Some(t1), <s> Tenv(Typed(e, t2), env))

tc(s) : Tenv(Typed(Alt(App(Constr(c), xs), t0, e),
                   TFun([t1], t2)), env) ->
        Alt(App(Constr(c), ys), Some(t1),
            <s> Tenv(Typed(e, t2), <conc> (env', env)))
        where <get-constructor-arg-types> (c, t1) => ts;
              <zip(\ (x,t) -> Typed(x,t)\ )>(xs, ts) => ys;
              <map(split(VarName, type))> ys => env'
```

Chapter 10

Simplification

10.1 WF-Auxiliary

```
module WF-Auxiliary
imports Haskell-Kernel

rules

  MkTFun  : (x, y) -> TFun(x, y)
  MkTFun1 : (x, y) -> TFun([x], y)
  MkApp   : (f, es) -> App(f, es)
  MkApp1  : (f, e) -> App(f, [e])

  new-tvar :
    x -> TVar(a) where new => a

  new-typed-var :
    t -> Typed(Var(x), t) where new => x

  Comp :
    (f, g) -> Abs(x, t, App(f, App(g, x)))
    where <type; Dom> g => t; new => x

  Identity :
    t -> Abs([Typed(Var(x), t)], Some(t), Typed(Var(x), t))
    where new => x

strategies

  value =
    rec x(Typed(Var(id) +
          Lit(id) +
          Constr(id), id) +
          Abs(id, id, id))
```

```
  linear =
    ?(x, t);
    <atmostonce(?Var(x))> t;
    <not(underabs(?x))> t

  underabs(s) = oncetd(App(id, Abs(id, id, oncetd(Var(s)))))
```

10.2 WF-Rules: Reduction Rules

```
module WF-Rules
imports Haskell-Lib WF-MapGen WF-Auxiliary
```

10.2.1 Abstraction and Application

Beta reduction. Rule `BetaOne` defines the application of a function to its first argument. Rule `Beta` reduces an application for as many arguments as possible, taking account of the fact that there may be fewer formal than actual parameters, or the other way around. The strategy `rest-zip` matches formal with actual parameters and produces the rest lists of formal parameters `ys` (empty in case of saturation), actual parameters `bs`, and a substitution `sbs` mapping formal to actual parameters. The rules only apply if for each argument either the actual parameter is a value or the formal parameter is linear in the body. Note that any empty abstraction or application will be cleaned up by the `Uncurry` rules.

```
rules

  BetaOne :
    App(Abs([x|xs], t, e), [a|as]) ->
    App(Abs(xs, t, <expsubst> ([x], [a], e)), as)
    where <value> a + <linear> (x, e)

  Beta :
    App(Abs(xs, t, e), as) ->
    App(Abs(ys, t, <expsubst> (sbs, e)), bs)
    where <rest-zip(id)> (xs, as) => (ys, bs, sbs);
          (<lzip((id,value) + (Fst,id); linear)> (sbs, e))
```

Extensionality

```
  Eta :
    Abs(xs, t, App(e, xs)) -> e
    where <lzip(not(in))> (xs, e)
```

Inlining

```
  Inl :
    Let([Valdef(Var(x), e1)], e2[Typed(Var(x),t)]) ->
    Let([Valdef(Var(x), e1)], e2[<etrename> e1])
```

Dead code elimination

```
Dead :
  Let([Valdef(Var(x), e1)], e2) -> e2
  where <not(in)> (Var(x), e2)
```

10.2.2 Type Abstraction and Type Application

```
TBeta :
  TInst(TAbs(as, e), ts) -> <tpsubst> (as, ts, e)

TEta :
  TAbs(as, TInst(e, as)) -> e
  where <lzip(not(in))> (as, e)

TBeta :
  TApp(Forall(as, t), ts) -> <tpsubst> (as, ts, t)
```

10.2.3 Case

Case specialization

```
CaseConstr :
  Case(Typed(Constr(c), t), as) -> e
  where <fetch(?Alt(Constr(c), _, e))> as

CaseConstr :
  Case(App(Typed(Constr(c), ct), es), as) ->
  <expsubst> (xs, es, e)
  where <fetch(?Alt(App(Constr(c), xs), t, e))> as;
        (<list(value)> es + <lzip(linear)> (xs, e))
```

Application distributes over case

```
CaseDistL :
  App(Case(e, as), es) -> Case(e, as')
  where <lzip(ArmAppL)> (as, es) => as'

ArmAppL :
  (Alt(c, t, e), es) -> Alt(c, t, App(e, es))

CaseDistR ::
  App(?f, split-fetch(?Case(e, as)); ?(es1, es2)) -->
  !Case(e, <lzip(AltAppR)> (as, (f, es1, es2)))

AltAppR :
  (Alt(c, t, e), (f, es1, es2)) ->
  Alt(c, t, App(f, <concat> [es1,[e],es2]))
```

10.2.4 Cata and Build

cata-build fusion

```
  CataBuild :
    App(Cata(t1, t2, fs), [Build(t1, g)]) ->
    App(TInst(g, [t2]), fs)
```

specialization of a cata applied to a constructor

```
  CataConstr :
    App(Cata(t1, t2, fs), [Typed(Constr(c), t')]) -> f
    where
      <(get-constructors, id);
       zipFetch(?(ConstrDecl(_,_,c,_), f))> (t1, fs)

  CataConstr :
    App(Cata(t1, t2, fs), [App(Typed(Constr(c), t'), es)]) ->
    App(f, <zip(MkApp1)> (fs', es))
    where
      <(get-constructors, id); zipFetch(?(ConstrDecl(_,_,c,_), f))
      >(t1, fs);
      <Ec> (t1, Cata(t1, t2, fs), c) => fs'
```

10.3 WF-Simplify

```
module WF-Simplify
imports WF-Rules Haskell-Normalize fixpoint-traversal
```

Simplification of expressions using the basic rules of the calculus.

```
strategies

  basic_rules =
    Beta + Eta + (Inl; Dead) + TEta + TBeta +
    CaseConstr + CaseDistL + CaseDistR + Uncurry

  basic-cata = CataConstr + CataBuild + basic_rules

  basic-tycon = basic_rules

  simplify = innermost(basic-cata)

  simplify' = innermost(basic-tycon)
```

Chapter 11

The Warm Fusion Transformation

11.1 WF-Main: Transforming all Definitions

```
module WF-Main
imports WF-Trans

strategies

  main = iowrap(topwrap(Main))

  topwrap(s) = Module(id, id, TopDecls(s)) + Program(TopDecls(s))

  Main = etrename;
         where(collect-data-defs);
         InitWF;
         repeat(TransformDecl <+ NormD);
         ExitWF

rules

  InitWF :
    ds -> ([], [], ds)

  ExitWF :
    (ds1, ds2, []) -> <reverse> ds2

  TransformDecl :
    (ds1, ds2, [d @ Valdef(Var(name),e) | ds3]) ->
    ([d' | ds1], [d' | ds2], ds3)
    where
      <debug(!"transforming: ")> name;
      <ior(inline(!ds1); say(!"  inlined"),
           Transform;    say(!"  transformed"))> d => d'
```

```
  NormD :
    (ds1, ds2, [d| ds3]) -> (ds1, [d| ds2], ds3)

strategies

  inline(mkenv) = manytd(Inline(mkenv)); simplify

rules

  Inline(mkenv) :
    Typed(Var(x), t) -> <tpsubst; etrename> (sbs, e)
    where mkenv; fetch(?Valdef(Var(x), e)); <not(in)> (Var(x), e);
          <tpunify> [(<type> e, t)] => sbs
```

11.2 WF-Trans: Transforming one Definition

```
module WF-Trans
imports WF-Rules WF-DynamicRules WF-CataIntro
        WF-Split WF-Simplify WF-Unfold
```

Strategy `Transform'` embodies the basic idea of the warm fusion transformation. First introduce the build-cata identity in the body of the function definition. Then split the body into a wrapper and a worker. Unfold the wrapper in the worker to obtain a worker that is recursive with respect to itself. Derive a catamorphism from the definition of the worker. Finally, unfold the transformed worker back in body of the wrapper. The intermediate results of this transformation are cleaned up by simplifying them.

```
strategies

  Transform' =
    IntroBuildCata;
    simplify;
    SplitBodyCP;
    Unfold1in2;
    [id, simplify;
         MakeCataBody];
    Unfold2in1;
    simplify
```

The transformation rule above succeeds if the function definition it is applied to is a function that consumes a data structure and produces a new one. The result will be a function definition of the form `Abs(...,Build(...Cata(...)...))`. The basic transformations that we have defined can also deal with functions that either consume *or* produce a data structure. A consumer will be transformed to a `Cata` and a producer to a `Build`. The definitions below factor out the basic transformations from the pipeline above into transformations that achieve the three kinds of transformation.

The transformation `Transform` tries all three transformations. First it tries to introduce a build and cata wrapping the body of the function. If that succeeds the function is at least a producer of a data structure and possibly a consumer as well. Otherwise it might only be a consumer.

```
Transform =
  ((IntroBuildCata;
    simplify;
    (ConsumerProducer
     <+ Producer
     <+ NonRecursiveProducer))
   <+ Consumer);
  simplify
```

A consumer/producer can be turned into `Cata` form by first splitting the body at the case expression and then transforming the split off definition with `BodyToCata`.

```
ConsumerProducer =
  SplitBodyCP;
  BodyToCata
```

The strategy `BodyToCata` takes the definitions of the wrapper and worker functions. It unfolds the wrapper in the worker, simplifies the result and then tries to fuse the worker with the copy function (`Cata(c1,...,cn)`). The result is unfolded in the wrapper to obtain the new definition of the function.

```
BodyToCata =
  Unfold1in2;
  [id, simplify;
       SplitBodyPall;
       Unfold1in2;
       [id, simplify;
            MakeCataBody];
       Unfold2in1];
   Unfold2in1
```

The body of a producer cannot be turned into a `Cata`. Therefore workers are split off to catch the static parameters of the function.

```
Producer =
  SplitBodyP;
  Unfold1in2;
  [id, simplify;
       SplitBodyP;
       Unfold1in2;
       [id, simplify];
       LetUnfold2in1];
   LetUnfold2in1
```

In case of a producer that is not recursive there is nothing to do after introducing the `build-cata` and simplifying.

```
NonRecursiveProducer =
  id
```

In the case of a consumer, i.e., a function that consumes a datastructure, but does not produce a new one, the build-cata introduction fails, but the rest of the transformation is the same as in the case of a consumer-producer.

```
  Consumer = ConsumerProducer
```

11.3 WF-CataIntro: Introducing Catamorphisms

```
module WF-CataIntro
imports Haskell-Build-Cata Haskell-Lib
rules

  MkBuildCata :
    e -> Build(t1, TAbs([t2], Abs(fs, Some(t2),
                                App(Cata(t1, t2, fs), [e]))))
    where new-tvar => t2; <type> e => t1;
          <get-constructors> t1 => cdecls;
          <lzip(AbsConstr)> (cdecls, (t1, t2)) => fs

  AbsConstr :
    (ConstrDecl(_, _, c, ts), (t1, t2)) -> Typed(Var(f), TFun(ts', t2))
    where new => f; <map(try(?t1;!t2))> ts => ts'

strategies

  IntroBuildCata = Valdef(id, under-abs(MkBuildCata))

  under-abs(s) = rec x((Abs(id, id, x) + TAbs(id, x)) <+ s)
```

11.4 WF-Split: Abstracting Expressions

```
module WF-Split
imports Haskell-Build-Cata Haskell-Lib
```

11.4.1 Function Parameters

The rule `AllParams` transforms a function definition into the list of formal parameters of the function.

```
rules

  AllParams : Valdef(Var(f), Abs(xs, t, e)) -> xs
```

The rule `call-args` recognizes a call site of a function `f` and transforms it to the list of arguments of the function.

```
  call-args(mkf) : App(Typed(Var(f), t), es) -> es where mkf => f
```

11.4.2 Non-static Function Parameters

The rule `NonStaticParams` derives the list of non-static parameters of a function by taking the list of all parameters and eliminating those that are passed verbatim to recursive calls of the function.

```
rules

  NonStaticParams :
    Valdef(Var(f), Abs(xs, t, e)) -> xs'
    where <collect(call-args(!f))> e => argss;
          <non-static> ([], xs, argss) => xs'

strategies

  non-static = repeat(NonStatic1 <+ NonStatic2 <+ NonStatic3)

rules

  NonStatic1 : (ys, [], _) -> <reverse> ys

  NonStatic2 : (ys, [xt @ Typed(Var(x), t) | xs], argss) ->
               (ys, xs, <map(Tl)> argss)
               where <map(?[xt | _])> argss

  NonStatic3 : (ys, [x | xs], argss) ->
               ([x | ys], xs, <map(try(Tl))> argss)
```

11.4.3 Abstraction of Expression

The rule `SplitExpr` split an expression `e` in a new function definition with `e` as body and a call to that function to replace the expression, i.e.,

```
e -> (f xs, f = \xs -> e)
```

The function abstracts over the variables in `mkxs`.

```
rules

  SplitExpr(mkxs) :
    e -> (App(Typed(Var(f), t), xs), Valdef(Var(f), body))
    where mkxs => xs; new => f;
          <etrename> Abs(xs, Some(<type> e), e) => body;
          <type> body => t
```

11.4.4 Split Wrapper

Given a strategy for splitting an expression in a call and a definition, the rule `SplitBody` splits the expression in the body of a function definition that sits in the argument of the `Build` expression under its leading value and type abstractions.

```
rules

  SplitBody(split) :
    Valdef(Var(x), body) -> [Valdef(Var(x), body'), def]
    where <under-abs-build(split => (e, def); !e)> body => body'

strategies

  under-abs-build(split) =
    rec x((Abs(id, id, x) + TAbs(id, x) + Build(id, split)) <+ split)
```

11.4.5 Reordering the Arguments

The rule `SplitCaseExpr` splits an expression just like `SplitExpr`, but puts the argument that is inspected by the case statement in the expression first in the list of variables that is abstracted over.

```
rules

  SplitCaseExpr(mkxs) :
    e -> <SplitExpr(!xs)> e
    where <ReorderArgs> (e, <mkxs>()) => xs

  ReorderArgs :
    (e, xs) -> [q | xs']
    where <casevar> e => q; <diff> (xs, [q]) => xs'

strategies

  casevar = oncetd(?Case(q @ Typed(Var(_),_),_)); !q
```

11.4.6 Split Combinations

The building blocks for splitting functions can be combined in various ways; `CP` stands for Consumer/Producer, `P` stands for Producer.

```
strategies

  SplitBodyCP =
    where(NonStaticParams => vs);
    SplitBody(SplitCaseExpr(!vs))

  SplitBodyCPall =
    where(AllParams => vs);
    SplitBody(SplitCaseExpr(!vs))

  SplitBodyP =
    where(NonStaticParams => vs);
    SplitBody(SplitExpr(!vs))
```

```
  SplitBodyPall =
    where(AllParams => vs);
    SplitBody(SplitExpr(!vs))
```

11.5 WF-DynamicRules: Implementing the Promotion Theorem

```
module WF-DynamicRules
imports Haskell-Lib
```

11.5.1 Generation of Dynamic Rules

```
rules

  DynRules :
    (t, g, c) -> (ys, zs, rls)
    where
      <Ec> (t, g, c) => es;
      <map(type; split(dom; new-typed-var, range; new-typed-var));
       unzip> es => (ys, zs);
      <zip(id)> (<zip(MkApp1; simplify)> (es, ys), zs) => rls
```

11.5.2 Application of Dynamic Rules

```
rules

  AppDynRule(mkrls) :
    App(f, y) -> z
    where <AppDynRule'(mkrls)> App(f, y) => z

  AppDynRule(mkrls) :
    Typed(y, t) -> z
    where <AppDynRule'(mkrls)> Typed(y, t) => z

  AppDynRule'(mkrls) : e -> e'
    where <lzipFetch(IsRule; ?e')> (<mkrls>(), e)

  IsRule :
    ((l, r), t) -> r
    where <equal( \ Typed(e, _) -> e \ )> [(l,t)]

strategies

  dsimplify(mkrls) = innermost(AppDynRule(mkrls) <+ basic_rules)
```

11.5.3 Construction of Function for Constructor

```
rules

  MkH :
    (ConstrDecl(_, _, c, ts), (g, e, t)) -> h
    where
      <DynRules> (t, g, c) => (ys, zs, rls);
      !Typed(Constr(c), TFun(ts, t)) => ct;
      <dsimplify(!rls)>
         Abs(zs, None, App(<etrename> e,[App(ct, ys)])) => h;

      // checking that all ys have been rewritten
      <not(oncetd({y : ?Var(y);
                      where(<fetch(Typed(Var(?y),id))> ys)});
           debug(!"MkH failed: ")) > h
```

11.5.4 Construction of Catamorphism

Construct cata by composition of body of worker with copy cata function

```
rules

  MakeCataBody :
    Valdef(Var(g), e) -> Valdef(Var(g), Cata(t1, t2, hs))
    where <type> e => tg;
          <split(dom, range)> tg => (t1, t2);
          <get-constructors> t1 => cdecls;
          <lzip(MkH)> (cdecls, (Typed(Var(g), tg), e, t1)) => hs
```

11.6 WF-MapGen: Generation of Maps from Data Types

```
module WF-MapGen
imports Haskell-Lib
```

Given the datatype constructor `T`, the strategy `E` generates the map function over `T`. `E` is applied to a pair `(env, t)` of an environment `env` and a type `t`. The environment maps types to functions to be applied to that type.

```
strategies

  E = rec x(E0 <+ E1 <+ E2(x))

rules

  E0    : (env, t) -> g
          where <lookup> (t, env) => g

  E1    : (env, TVar(a)) -> <Identity> TVar(a)
```

```
  E1    : (env, TCon(a)) -> <Identity> TCon(a)

  E1    : (env, TApp(tcon, ts)) -> <Identity> TApp(tcon, ts)
          where <not(lzipFetch(lookup))> (ts, env)

  E2(s) : (env, TApp(tcon, ts)) ->
          App(<MkMapBody> (TApp(tcon, ts), rs), fs)
          where <lzipFetch(lookup)> (ts, env);
                // this might miss deeper embedded recursion?
                <rzip(s)> (env, ts) => fs;
                <map(type; range)> fs => rs

rules

  Ec : (t, g, c) ->
       <(id, get-constructor-arg-types); rzip(E)> ([(t, g)], (c, t))
```

11.6.1 Generating Map Functions

The rule `MkMapBody` generates the implementation of a map function, mapping a `d as` value to a `d bs` value. The implementation is in terms of build and cata.

```
  (d as, bs) ->
  \(f1 :: a1 -> b1) ... (fn :: an -> bn) :: (d as -> d bs) ->
   build[d bs](/\ a -> \c1 ... cn -> cata[d as][a](g1,...,gn))
```

The `gi` are functions that apply the parameter functions `f` in the appropriate way. For lists we get

```
  (List b, b') ->
  \(f :: b -> b') :: (List b -> List b') ->
  build[List b'](/\ a -> \c1 c2 ->
                 cata[List b][a](c1, \ x xs -> c2(f x)(xs)))

rules

  MkMapBody :
    (TApp(tcon, ts), ts') ->
    Abs(fs, Some(TFun([TApp(tcon, ts)], TApp(tcon, ts'))),
        Build(TApp(tcon, ts'),
              TAbs(a, Abs(cs, Some(a), Cata(TApp(tcon, ts), a, gs)))))
    where
      new-tvar => a;
      <zip(MkTFun1; new-typed-var)> (ts, ts') => fs;
      <zip(id)> (ts, fs) => env0;
      ![(TApp(tcon, ts), <Identity> a) | env0] => env;
      <get-constructors> TApp(tcon, ts) => cdecls;
      <lzip(MkG); unzip> (cdecls, (env, a)) => (cs, gs)

  MkG :
```

```
(ConstrDecl(_, _, c, ts), (env, res)) ->
(f, Abs(xs, Some(res), App(f, <zip(MkApp1)> (hs, xs))))
where <rzip(E)> (env, ts) => hs;
      <map(type; split(dom, range)); unzip> hs => (doms, rans);
      <map(new-typed-var)> doms => xs;
      <new-typed-var> TFun(rans, res) => f
```

Bibliography

[1] www.stratego-language.org.

[2] www.cs.uu.nl/people/visser/sdf2/.

[3] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software—Practice & Experience*, 30:259–291, 2000.

[5] Mark G. J. Van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996.

[6] R. M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[7] W. N. Chin. Safe fusion of functional expressions. *ACM Lisp Pointers*, 5(1):11–20, 1992. Proceedings of the 1992 ACM Conference on Lisp and Functional Programming.

[8] M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Twente University, 1992.

[9] Pascal Fradet and Daniel Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, January 1991.

[10] Andrew John Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, January 1996.

[11] Andy Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 223–232. ACM Press, 1993.

[12] T. Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[13] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. *ACM SIGPLAN Notices*, 31(6):73–82, May 1996. Proceedings of the International Conference on Functional Programming (ICFP'96), Philadelphia.

[14] Patricia Johann. An implementation of warm fusion. Available at ftp://ftp.cse.ogi.edu/pub/pacsoft/wf/, 1997.

[15] Merijn de Jonge. A pretty-printer for every occasion. In Ian Ferguson, Jonathan Gray, and Louise Scott, editors, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, Limerick, Ireland, June 2000. Technical report, University of Wollongong, Australia.

[16] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In S. L. Peyton Jones, editor, *Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 314–323. ACM Press, June 1995.

[17] 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.

[18] G. Malcolm. Homomorphisms and promotability. In *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 335–347. Springer-Verlag, 1989.

[19] G. J. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, August 1990.

[20] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In R. J. M. Hughes, editor, *Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, August 1991.

[21] László Németh. *Catamorphism Based Program Transformation for Non-Strict Functional Langauges.* PhD thesis, University of Glasgow, 2000.

[22] Will Partain. The `nofib` benchmark suite of haskell programs. In J. Launchbury and P. M. Sansom, editors, *Functional Programming*, pages 195–202. Springer-Verlag, 1992.

[23] Simon Peyton Jones, John Hughes, et al. Report of the programming language Haskell98. a non-strict, purely functional language, February 1999.

[24] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In R. J. M. Hughes, editor, *Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666. Springer-Verlag, September 1991.

[25] Simon L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.

[26] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In Arvind, editor, *Functional Programming and Computer Architecture (FPCA'93)*, pages 233–242, Copenhagen, Denmark, 1993. ACM Press.

[27] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In S. L. Peyton-Jones, editor, *Functional Programming and Computer Architecture (FPCA'95)*, San Diego, California, June 1995.

[28] Andrew Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[29] V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–326, 1986.

[30] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

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

[32] 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.

[33] Eelco Visser. *The Stratego Library*. Institute of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 1999.

[34] Eelco Visser. Language independent traversals for program transformation. In Johan Jeuring, editor, *Workshop on Generic Programming (WGP2000)*, Ponte de Lima, Portugal, July 6, 2000. Technical Report UU-CS-2000-19, Universiteit Utrecht.

[35] 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).

[36] 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).

[37] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.