

Type-safe Functional Strategies

Ralf Lämmel* Joost Visser*

Abstract

We demonstrate how the concept of strategies originating from term rewriting can be introduced in a typed, functional setting. We provide a model of strategies based on a further generalisation of updatable, monadic, generalised fold algebras. We show how strategies can be used as a structuring device for functional programming.

1 Introduction

Strategies have been proposed to control term rewriting, i.e., to describe when and how one-step term rewrite rules are to be applied [5, 1, 10]. For instance, strategies may define term traversal schemes independently of the rules applied during traversal. Such traversal strategies can describe term traversals as required for constructing program transformations and program analyses.

The strategy language *Stratego* [10] is the most ambitious and expressive implementation of the idea of strategies. Stratego's primitive strategies and strategy definition constructs enable the definition of arbitrary traversal strategies for terms of any shape in a first-order setting. Stratego is powerful and flexible, but it is (as yet) *untyped*. As a consequence, strategic rewriting offers low static safety and few theorems for free. Furthermore, the strategic programs are often hard to understand because types are not available for an abstract, non-operational understanding of program elements. Typing a strategy language like Stratego is not straightforward as we will discuss later.

Although term rewriting and functional programming are close relatives, traversal strategies have not been investigated in full generality in the context of functional programming. It is common practice to encode traversals either by general recursion and pattern matching, or by folds [6]. In this paper, we

*CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands. Email: `ralf@cwi.nl` and `Joost.Visser@cwi.nl`.

take up the challenge of providing a *typed, functional* model of strategies. We will provide functional counterparts of the primitives of the strategy language Stratego, and we will show how functional strategic programming proceeds from these primitives. The key of our approach is to model strategies using a generalisation of *updatable fold algebras* [4].

Strategies extend the set of structuring techniques at the disposal of a functional programmer in the following way. On the one hand, *folds* are known to provide the device to structure programs after the values they consume [6]. Actually, the fold traversal scheme can be seen as an example of a fixed strategy parameterised by the one-step rewrite rules to be applied to the nodes in bottom-up manner. On the other hand, *monads* help to structure programs after the values they compute [11]. In this way, folds and monads complement each other [7]. Strategies take this development one step further. Strategies allow parts of programs to be structured *independently* of the values that are consumed or computed, and then to specialise these generic, reusable program parts to implement specific functionality.

The paper is structured as follows. Section 2 introduces the concept of a strategy, as featured in strategic rewriting, and discusses the lack of types in strategy languages such as Stratego. Section 3 explains the notion of updatable fold algebras, that we intend to use as a basis for typing strategies. Section 4 develops the typed functional model of strategies involving a generalisation of fold algebras. Section 5 shows functional strategies in action.

2 Strategies

In this section we set the scene by introducing the concept of strategies, as featured in strategic rewriting. We briefly explain the strategy language Stratego to indicate how strategies are used in programming.

Strategic rewriting In ordinary term rewriting, a programmer specifies rewrite rules which are applied to an input term according to a fixed normalisation strategy, such as innermost rewriting. In *strategic* rewriting, the programmer is enabled to specify traversal strategies explicitly by composing constant strategies with strategy combinators. In this paradigm, rewrite rules are examples of constant strategies, and traversal schemes (like ‘innermost’) are unary strategy combinators.

Stratego is a programming language for strategic rewriting. Stratego offers a small set of primitive strategies and strategy definition constructs which enable the definition of arbitrary traversal strategies. Figure 1 lists the most important primitive strategies, as well as some prime examples of defined ones. For instance, the defined strategy combinator `bottomup` applies

Primitive	Meaning
<code>id</code>	do nothing
<code>fail</code>	always fail
<code>?t</code>	match
<code>!t</code>	build
<code>one(s)</code>	apply <code>s</code> to exactly one child
<code>some(s)</code>	apply <code>s</code> to at least one child
<code>all(s)</code>	apply <code>s</code> to every child
<code>s ; s'</code>	sequential composition
<code>s <+ s'</code>	choice
<code>rec x(s)</code>	recursive closure
<code>try(s)</code>	= <code>s <+ id</code>
<code>repeat(s)</code>	= <code>rec x(try(s;x))</code>
<code>bottomup(s)</code>	= <code>rec x(all(x);s)</code>
<code>oncebu(s)</code>	= <code>rec x(one(x)<+s)</code>
<code>innermost(s)</code>	= <code>repeat(oncebu(s))</code>

Figure 1: Primitives and defined strategies in the Stratego language.

its argument strategy `s` once to every subterm in bottom-up fashion.

Rewrite rules in Stratego are actually specific instances of strategies where a match strategy is followed by a build strategy. For such specific strategies special syntactic sugar is available. The following are equivalent:

rule: `r : t1 -> t2`
strategy: `r = ?t1 ; !t2`

Here, `t1` and `t2` are arbitrary terms, possibly containing variables.

To apply a strategy `traverse` that models a traversal strategy to a strategy `rewrite` that models a rewrite rule, the latter is simply supplied to the former as a parameter, as in `traverse(rewrite)`. To apply a strategy `s` to a term `t`, the idiom `<s>t` is used.

A simple example of a Stratego program, adapted from [9], is depicted in Figure 2. The program defines a transformation for a small lambda language that removes some dead code. The rewrite rules `Eta` and `CFE` perform eta-reduction and constant function elimination, respectively. These two rules are combined with the choice operator and fed into the traversal defined with `bottomup` and `repeat` of Figure 1. The strategies `fvars` and `in` are not shown. They perform free variable analysis and a membership test, respectively. We will use this example throughout the paper.

The benefits of strategies Strategies give the programmer complete control over traversals. Moreover, these traversal strategies can be defined separately, instead of tangling them with one-step rewrite rules. Strategies can

```

signature
constructors
  TVar   : Name           -> Type
  Arrow  : Type * Type    -> Type
  Var    : Name           -> Expr
  Apply  : Expr * Expr    -> Expr
  Lambda : Name * Type * Expr -> Expr
rules
  Eta : Lambda(x,t,Apply(e,Var(x))) -> e where <not(in)>(x, <fvars> e)
  CFE : Apply(Lambda(x,t,e1),e2)-> e1 where <not(in)>(x, <fvars> e1)
strategies
  remove-dead = bottomup(repeat(Eta <+ CFE))

```

Figure 2: An example Stratego program.

be programmed generically, in the sense that they are defined once and for all for terms of any shape. By appropriate instantiations and refinements these strategies can be specialised to construct specific programs. All these properties are demonstrated by the example in Figure 2. Thus, strategies allow a mixture of generic and specific programming, leading to concise programs where generic, reusable parts are defined independently of the parts that are specific for a certain term language and a certain problem. See [9, 10] for a more elaborate account of strategic programming.

Types are sacrificed Unfortunately, Stratego offers strategies at the expense of types. Stratego is an untyped language (though the current implementation offers some checks on the arities of term constructors), and providing a type-system for it is not straightforward, as argued in [9].

A first-order many-sorted type system as for ordinary rewriting is insufficient for several reasons. In ordinary rewriting, all rules are required to be type-preserving. Non-type-preserving traversals—which are needed for scenarios such as interpretation, program analysis and term reduction—are usually encoded using an auxiliary outermost function symbol. But strategic rewriting is all about avoiding such tangled encodings, and thus non-type-preserving rules and traversals are essential. If a restriction of type-preservation would be imposed, the first-order many-sorted rule type could be lifted to the strategy level, as exemplified for ELAN in [1]. Without this restriction, however, a more elaborate type system is needed, especially to cover generic primitives such as Stratego’s `one`, `some`, and `all`.

A type system that improves static safety of strategic programs should enforce the following restrictions. Firstly, when a strategy is applied to a well-formed term (a term that is typable), the resulting rewritten term should be well-formed as well. This gives a concept of well-typedness of strategies as a lifting of well-typedness of terms (rather than of type-preserving rewrite

rules). Secondly, a parameterised strategy may require its argument strategies to be of certain types, in order to be typable itself.

Types are important for static safety, but also from a program comprehension and documentation perspective. Furthermore, types can be beneficial for modularity, abstraction, separate compilation, and optimisation. Of course, it is of prime importance that the benefits of strategic programming, such as its opportunities to mix generic and specific programming, are not cancelled when types are given to strategies.

3 Updatable fold algebras

Folds [6] are widely used in functional programming. For instance, the *Haskell Prelude* module contains the function `foldr` for the datatype of lists. In [7] the notion of *generalised* folds—pertaining to a system of datatypes, rather than a single datatype—was combined with the notion of monads. In [4] we improved on these monadic generalised folds by making them *updatable*. Here we made essential use of the possibility of passing fold ingredients to a fold function by means of a fold *algebra*. The functional model of strategies to be presented in Section 4 will be based on a slight generalisation of updatable monadic fold algebras. For this reason we will first recapitulate this notion, and then generalise it.

Remark In the examples throughout the paper, we use *Haskell 98* extended with multi-parameter type classes, and functional dependencies.

Fold algebras Figure 3 shows an example system of two datatypes `Type` and `Expr`, together with its associated fold algebra type `CataAlg`, and two examples of derivable basic fold algebras: `idmap` and `crush`. For concision, we only show the code relevant for the constructors `Var` and `Apply`. Note that the algebra type is parameterised with a monad, and that its constituents are *monadic* functions. We consider monadic fold algebras to cope with effects such as propagation and accumulation involved in traversals. In the functional model of strategies, we will make use of the backtracking effect to model failure and success of strategies. In [4] non-monadic updatable folds are considered in combination with monadic ones. The algebra type is further parameterised by the result types of the traversal, one for each datatype. By suitable instantiations of the fold algebra parameters, the algebra types for type-preserving and type-unifying algebras are obtained. These are captured by the type synonyms `PreservingCataAlg` and `UnifyingCataAlg`. An example is the type-preserving basic fold algebra `idmap`, which models the traversal that replaces each constructor by itself. The basic fold algebra

```

data Type = TVar String | Arrow Type Type
data Expr = Var String | Apply Expr Expr | Lambda String Type Expr

data Monad m => CataAlg m e t =
  CataAlg{ var    :: String -> m e
          , apply :: e -> e -> m e
          , lambda :: String -> t -> e -> m e
          , tvar   :: String -> m t
          , arrow  :: t -> t -> m t }
type PreservingCataAlg m = CataAlg m Expr Type
type UnifyingCataAlg m a = CataAlg m a a

idmap :: Monad m => PreservingCataAlg m
idmap = CataAlg{ var    = \x    -> return (Var x)
               , apply = \f a  -> return (Apply f a)
               , ... }

crush :: (Monad m, Monoid a) => UnifyingCataAlg m a
crush = CataAlg{ var    = \x    -> return mempty
               , apply = \f a  -> return (f 'mappend' a)
               , ... }

class FoldFunction alg t t' | alg t -> t' where
  fold :: alg -> t -> t'
instance Monad m => FoldFunction (CataAlg m e t) Expr (m e) where
  fold alg (Var x) = var alg x
  fold alg (Apply f a) = do f' <- fold alg f
                           a' <- fold alg a
                           apply alg f' a'
  ...

```

Figure 3: Updatable fold algebras.

`crush` models the type-unifying traversal that replaces leaf constructors with the empty element of a monoid, and combines the result of processing the children of non-leaf nodes with the binary operator `mappend` of the monoid.

Finally, Figure 3 shows the fold function which is parameterised by a fold algebra. The function performs a bottom-up traversal of its argument term and replaces all constructors by the appropriate functions from the algebra. For any given system of mutually recursive datatypes, all elements listed in Figure 3 can be derived in a straightforward fashion.

Programming with updatable fold algebras Programming with updatable fold algebras proceeds along the following steps: (i) write a *fold algebra update*, (ii) apply the update to a basic fold algebra, and (iii) feed the updated fold algebra to a fold function. A small example of programming with updatable folds is given in Figure 4. The function `fvars` performs a

```

fvars :: Expr -> Id [String]
fvars = fold (fvUpd crush)
fvUpd alg = alg{ var    = \x -> return [x]
                , lambda = \x t b -> return (filter (x/=) b) }

```

Figure 4: An example of programming with updatable folds.

```

data Alg e t e' t' =
  Alg{ var      :: String      -> e'
      , apply   :: e -> e      -> e'
      , lambda  :: String -> t -> e -> e'
      , tvar    :: String      -> t'
      , arrow   :: t -> t      -> t' }
type CataAlg m e t = Alg e t (m e) (m t)

```

Figure 5: Generalised algebra type.

free variable analysis by updating the basic algebra `crush` for the constructors `Var` and `Lambda`. The resulting algebra is fed to the function `fold`. In this case, the monad parameter is instantiated with the identity monad `Id`. Note that although `fvars` is perfectly typable in a functional setting, it is not type-preserving from a rewriting perspective.

The basic fold algebras capture *generic* traversal. This generic behaviour is refined with fold algebra updates that capture behaviour specific for certain constructors of a particular datatype system. As we will show in Section 5, functional strategies allow an even further degree of genericity.

Generalised algebras In order to be able to model strategies, fold algebras need to be generalised. These generalised algebras have *two* type parameters for each datatype in the system: one for the occurrences of the type in argument positions, and one for the occurrences in result positions. As will become clear below, this generalisation is needed because traversal strategies, unlike folds, need not recurse into children. Consequently, the argument types (corresponding to the children) may be instantiated differently than the result type (corresponding to the node itself). For the system of datatypes of Figure 3, the generalised algebra type is shown in Figure 5. The original more specific fold algebra type `CataAlg` can be reconstructed from the generalised algebra type, as indicated by the type synonym.

4 A typed model of strategies

We start this section with a functional reformulation of the Stratego example given in Section 2. The reformulation gives the reader a first indication of the functional model of strategies that we will present. Then the model itself

```

remove_dead :: PreservingAlg Maybe
remove_dead = bottomup (repeat (eta 'choiceS' cfe))
eta, cfe :: PreservingAlg Maybe
eta = failS{ lambda = \x t b -> case b of
    (Apply e (Var y))
        -> do fvs <- fvars<>e
            guard ((x==y) && (not (x 'elem' fvs)))
            return e
    _ -> mzero }
cfe = failS{ apply = \f a -> case f of
    (Lambda x t b)
        -> do fvs <- fvars<>b
            guard (not (x 'elem' fvs))
            return b
    _ -> mzero }

```

Figure 6: Functional reformulation of the strategy for dead code elimination.

is developed in two steps. Firstly, we discuss some essential properties of strategies, and show how these can be obtained in a functional model via the generalised notion of algebras. Secondly, we will discuss how each of Stratego’s primitives can be recreated in the strategy-as-algebra model. The actual use of the model will be demonstrated in Section 5.

A functional reformulation of the example Figure 6 shows the functional formulation of the strategy for dead code elimination of Figure 2. The functional `remove_dead` strategy faithfully mimics the implementation in Stratego. The prime difference is that it has a type. To be exact, it is typed as a type-preserving algebra. The functional strategies `eta` and `cfe` mimic the rewrite rules of the Stratego example. Their types are identical to the type of `remove_dead`. They are defined by updating the strategy `failS`. They deviate from the Stratego counterparts in the following details. Firstly, the outer pattern-match in the Stratego rewrite rules is not part of the algebra updates. This match is done by the function to which the algebra will be fed, that is the strategy application operator `<>`. Secondly, the `where` clause of Stratego is a monadic `guard` in the functional strategy. Finally, in Stratego a non-linear match is used for the variable `x` in the pattern `Lambda(x,t,Apply(e,Var(x)))`, while the functional strategy performs an explicit equality check with `x==y`.

Strategies as algebras As the example shows, we model strategies as algebras. Within this model one-step rewrite rules are algebra updates and strategy combinators are algebra combinators. Depending on the instantiation of the parameters of the algebra type, the strategy modelled by an algebra can be type-unifying, type-preserving, or it can have a more general

```

type StrategyAlg m e t      = Alg Expr Type (m e) (m t)
type PreservingAlg m       = StrategyAlg m Expr Type
type PreservingCataAlg m   = PreservingAlg m
type PreservingStrategyAlg m = PreservingAlg m
type UnifyingAlg m e t a   = Alg e t (m a) (m a)
type UnifyingCataAlg m a   = UnifyingAlg m a a a
type UnifyingStrategyAlg m a = UnifyingAlg m Expr Type a

```

Figure 7: Instances of the generalised algebra type.

```

class Apply s t a | s t -> a where
  (<>) :: s -> t -> a
instance Apply (StrategyAlg m e t) Expr (m e) where
  s <> (Var x)      = var s x
  s <> (Apply a b) = apply s a b
  ...

```

Figure 8: Typed strategy application function.

type. The various type schemes which are relevant for the typed model of strategies are shown in Figure 7. For convenience, the more restrictive types for fold algebras are also reconstructed.

In Section 3, fold algebras were fed to a fold function to obtain actual traversal functions. Likewise, in our functional model of strategies, generalised algebras will be fed to a strategy application function. For this function we will use the infix operator `s<>t` corresponding to Stratego’s strategy application idiom `s<t>`. Figure 8 provides the definition for our example system. Note the type of an algebra to be passed to `<>`. The type `StrategyAlg` is an instance of the general algebra type, where the argument type parameter for each datatype is instantiated to the datatype itself. The operator `<>` is indeed not more general because it does not recurse into the children of the given term, and thus cannot change their type. We will see that some algebra combinators can be defined for a more general type than `StrategyAlg`. Figure 7 also derives the type of type-preserving and type-unifying strategy algebras. As the type synonyms indicate, the type of type-preserving strategy algebras coincides with the type of type-preserving fold algebras. The type-unifying strategy algebras and the type-unifying fold algebras are incomparable instances of the more general type `UnifyingAlg`.

The essence of strategies Note that in the strategy-as-algebra model, the essential properties of strategies are captured. The most obvious property is that a strategy can be applied to a term to obtain a new term. This is covered by the application function `<>`.

Another property of strategies is that they can be successful, and actually yield a term, or they can fail, in which case local backtracking scoped by the

```

class IdS s where
  idS    :: s
instance Monad m => IdS (PreservingAlg m) where
  idS = Alg{ var      = \x -> return (Var x)
            , apply   = \a b -> return (Apply a b)
            , ... }

class FailS s where
  failS  :: s
instance MonadPlus m => FailS (Alg e t (m e') (m t')) where
  failS = Alg{ var      = \x -> mzero
            , apply   = \a b -> mzero
            , ... }

```

Figure 9: Identity and fail strategies in the functional model.

choice operator (i.e., `<+` in Stratego) [10] may be needed. This property is captured by the monad in the algebra, which can be instantiated to a monad with `mzero` (encoding failure) and `mplus` (to compose success/failure values). The `Maybe` monad with `Nothing` denoting failure is well suited for that purpose. The `mplus` of the `Maybe` monad is biased to choose successful values from the left which is in accordance to Stratego’s primitive `<+`.

Strategies in Stratego have a number of convenient built-in effects: binding of variables, new variable generation, input and output capabilities. These properties can be obtained by stacking environment, state, and IO monads on the monad in the algebra.

Functional counterparts of Stratego’s primitives The definitions of the Stratego primitives (cf. Figure 1) in the functional model are discussed in the sequel. Since our primitives need to be typed, we can not be faithful to Stratego’s primitives in every detail. Note also that although our primitives embody generic concepts, we can not give generic definitions for them in *Haskell*. Instead, we present instances of these primitives for our example system of datatypes. The derivation of these primitives for a particular system is straightforward. Proper language support for generic definitions of the primitives will be discussed in the conclusions.

The functional counterparts of Stratego’s `id` and `fail` primitives are shown in Figure 9. The identity strategy corresponds to the type-preserving fold algebra `idmap` discussed in Section 3. The `fail` strategy is defined as an algebra that only contains constant functions returning `mzero`. Its type is not restricted.

Figure 10 shows two functional variants of the `all` primitive. They complement each other. The first variant closely mimics Stratego’s `all`, but it is restricted to type-preserving algebras. Its type cannot be generalised

```

class AllS s where
  allS  :: s -> s
instance Monad m => AllS (PreservingAlg m) where
  allS s = Alg{ var    = \x -> return (Var x)
               , apply = \a b -> do a' <- s<>a
                                   b' <- s<>b
                                   return (Apply a' b')
               , ... }
instance (Monoid a, Monad m) => AllS (UnifyingStrategyAlg m a) where
  allS s = Alg{ var = \x -> return mempty
               , apply = \a b -> do a' <- s<>a
                                   b' <- s<>b
                                   return (a' 'mappend' b')
               , ... }

```

Figure 10: The all strategy in the functional model.

```

class Sequence s s' s'' | s s' -> s'' where
  seqS :: s -> s' -> s''
instance MonadPlus m =>
  Sequence (Alg e t (m Expr) (m Type))
           (Alg Expr Type (m e') (m t'))
           (Alg e t (m e') (m t')) where
  seqS s s' = Alg{ var    = \x -> (var s x) >>= (s'<>)
                  , apply = \a b -> (apply s a b) >>= (s'<>)
                  , ... }

class Choice s where
  choiceS  :: s -> s -> s
instance MonadPlus m => Choice (Alg e t (m e') (m t')) where
  choiceS s s' = Alg{ var    = \x -> (var s x) 'mplus' (var s' x)
                    , apply = \a b -> (apply s a b) 'mplus' (apply s' a b)
                    , ... }

```

Figure 11: Sequence and choice in the functional model.

any further. The second variant is applicable to type-unifying strategies. It unifies the result types and, unlike the first variant (and Stratego's `all`), does not preserve the top node. Instead it combines the intermediate results obtained from the children using the binary operator `mappend` of a monoid. This primitive essentially relies on the generalisation of the algebra type. The functional counterparts for `one` and `some` can be derived in a similar way as for `all`.

For the recursive closure operator of Stratego no functional counterpart is needed, since recursive strategies are encoded as recursive functions. The functional counterparts of choice and sequential composition are shown in Figure 11. Note that the sequence of two strategies is a strategy that gets its

```

class Build s a | s -> a where
  build :: a -> s
instance Monad m => Build (UnifyingAlg m e t a) (m a) where
  build term = Alg{ var      = \x -> term
                  , apply   = \a b -> term
                  , ... }

```

Figure 12: The build primitive in the functional model.

```

class Comb a1 a2 a s1 s2 s | s -> a where
  comb :: (a1->a2->a) -> s1 -> s2 -> s
instance Monad m
=> Comb a1 a2 a (UnifyingAlg m e t a1)
              (UnifyingAlg m e t a2)
              (UnifyingAlg m e t a) where
  comb o s1 s2
    = Alg{ var = \x -> (var s1 x) 'mo' (var s2 x)
          , apply = \a b -> (apply s1 a b) 'mo' (apply s2 a b)
          , ... }

```

Figure 13: An additional primitive for combining type-unifying strategies.

argument types from the first strategy, and its result types from the second strategy. Also, the result types of the first strategy are the same as the argument types of the second. Thus, this primitive also essentially relies on the algebra generalisation. The choice primitive takes two strategies of the same, unrestricted type into a strategy of this same type. It uses the monadic `mplus` operator to combine the monadic algebra members for each constructor.

Figure 12 shows the functional build primitive (`!t` in Figure 1). It takes a term `term` as argument. The application of this strategy means to discard the traversed term and to return `term`. Note that all argument types are unified by this primitive. There is no functional counterpart for the match primitive, because we rely on the matching in the strategy apply function `<>` for matching the top node of a term, and on case expressions for matches on children. These functional style matches do not cover the usage `?x` of Stratego’s match primitive with `x` being a variable.

Additional primitives There is room for a few additional primitives in our functional model in order to cover usage patterns of strategic programs without relying on an untyped setting. For brevity, we only consider one example. In programming with untyped strategies, we can encode a strategy which applies two strategies to the input term, and then combines their results as follows:

```

comb(o,s1,s2) = ?x;!(<s1>x,<s2>x);o

```

```

try          :: (Choice a, IdS a) => a -> a
try s       = s 'choiceS' idS

repeat      :: (Choice s, IdS s, Sequence s s s) => s -> s
repeat s    = try (s 'seqS' (repeat s))

bottomup    :: (Sequence s s s, AllS s) => s -> s
bottomup s  = (allS (bottomup s)) 'seqS' s

oncebu, oncedt :: (Choice s, OneS s) => s -> s
oncebu s    = (oneS (oncebu s)) 'choiceS' s
oncedt s    = (s 'choiceS' (oneS (oncedt s)))

innermost   :: (Choice s, IdS s, Sequence s s s, OneS s) => s -> s
innermost s = repeat (oncebu s)

```

Figure 14: Generic functional strategies.

Note that this strategy encodes the intermediate results as a pair and assumes that `o` acts on a pair. Also, the strategy makes use of the match primitive with a variable as the pattern. We can abstract from this encoding by defining a primitive combinator that takes two strategies that unify result types, and combines these into a single strategy by applying a binary operator. In Figure 13, the corresponding functional primitive is shown.

5 Programming with functional strategies

Given the functional model of strategies presented in the previous section, functional strategic programming can commence. We start by showing how the defined generic strategies of Figure 1 can be constructed from the functional strategy primitives. Then we show how such generic strategies can be combined and refined to construct a specific traversal function. To this end, we return to the running example.

Defined functional strategies Figure 14 shows the typed counterparts of the untyped Stratego strategies defined in Figure 1. Like the untyped counterparts, these functional strategies are generic, in the sense that they are defined once and for all for any system of datatypes. The use of *Haskell* classes isolates these definitions from the non-genericity of the functional counterparts of the strategy primitives.

Free variable analysis In the example in Figure 2, the strategies `in` and `fvars` are used, but not defined. Instead of `in` we used the `Prelude` function

```

free_vars :: (Monad m, AllS s, Choice s, Build s (m [String]),
             Comb [String] [String] [String] s s s) => s -> s -> s
free_vars getvars boundvars
  = fv where fv = getvars 'choiceS'
              ( comb diff (allS fv)
                (boundvars 'choiceS' (build (return []))) )

lVars,lBnd :: MonadPlus m => UnifyingStrategyAlg m [String]
lVars = failS{ var    = \x    -> return [x] }
lBnd  = failS{ lambda = \x t b -> return [x] }

fvars :: MonadPlus m => UnifyingStrategyAlg m [String]
fvars = free_vars lVars lBnd

```

Figure 15: Free variables analysis.

`elem` for membership test in our functional reformulation in Figure 6. We will now discuss the definition of `fvars` which performs free variable analysis.

The strategic functional definition of `fvars` is given in Figure 15. Note that free variable analysis requires a type-unifying strategy. The functional strategy definition (which mimics the untyped solution given in [9]) is structured as a generic strategy for free variable analysis `free_vars`, which is parameterised with strategies `getvars` and `boundvars`. These fetch used variables and bound variables from nodes, respectively. For non-binding constructs (first alternative of choice), used variables are returned. For binding constructs (second alternative of choice) the bound variables are subtracted using `diff` from free variables of the children. For the particular system of datatypes consisting of `Type` and `Expr`, the strategy parameters are instantiated with the rules `lVars` and `lBnd`. All functionality specific to this datatype is concentrated in these rules, while `free_vars` is completely generic.

Strategies allow more genericity The usual approach to implement a function like free variable analysis in a functional language is by pattern-matching and explicit recursion. This approach is fully specific to the lambda language and the free variable problem. In the naive fold approach, pattern-matching and recursion are hidden in a generic, reusable fold function to which a fold algebra is fed that contains the one-step rules specific to free variable analysis for lambda terms. The approach with updatable folds (see Figure 4) employs a basic fold algebra `crush`, which separates generic type-unifying traversal behaviour from the language-specific and problem-specific functionality in the algebra updates.

As Figure 15 shows, the approach with functional strategies takes yet another step on the ladder of genericity. Free variable analysis is a problem that reoccurs for every language with binding constructs. Strategic programming

allows a generic formulation of free variable analysis that can be instantiated with rules that embody language-specific information on binding and bound variable positions.

6 Concluding remarks

Contribution On the one hand, providing a typed model for strategies in a functional language adds conceptual clarity to the area of programming with strategies. On the other hand, our model enriches functional programming with a new structuring technique, allowing more genericity. Programming with functional strategies is profitable in application areas such as language processing, program transformation, and program analysis as shown in [9]. Note that functional strategies go beyond first-order term rewriting strategies in several ways. For instance, the application of a traversal strategy might return functions instead of first-order terms. This feature can be used, for example, to encode multi-parameter traversals such as binary relations on terms.

Language support for strategies To define the primitive strategies such as `<>`, `seqS` and `choiceS` generically, the underlying notion of algebras is essential. In [4], we indicate how to define the slightly simpler notion of fold algebras in a generic setting. By contrast, existing generic languages such as PolyP [2] only support inductive definitions on the structure of datatypes, but not a constructor-sensitive notion of algebra for a given system of datatypes. Also, the kind of updating required for the customization of generic traversal strategies to accommodate specific functionality cannot be expressed. Updatable fold algebras—but not (yet) full functional strategies—are supported by the generator tool *Tabaluga*¹ for *Haskell*.

Limitations and perspective The restrictions imposed on strategies by types have the effect that certain things can no longer be expressed. In many cases, this effect is desirable, for instance, when unsafe programs are made inexpressible. In some cases, types might be more restrictive than desirable. We have no obvious and simple examples of such undesirable restrictions. However, we are aware of a fairly complex scenario for which our typed model is too restrictive. In this scenario, the constructors of a given signature are reused to build terms of a different but related signature (in the sense of a projection or homomorphism). This is useful for instance to describe abstract interpretations. Such constructor reuse is not possible in the model we have presented.

¹ *Tabaluga* homepage: <http://www.science.uva.nl/~kort/tabaluga>

References

- [1] P. Borovansky, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting. In Meseguer [8].
- [2] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
- [3] J. Jeuring, editor. *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, July 2000.
- [4] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In Jeuring [3], pages 46–59. available at <http://www.cwi.nl/~ralf/>.
- [5] S. E. M. Clavel, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [8].
- [6] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. FPCA'91*, volume 523 of *LNCS*. Springer-Verlag, 1991.
- [7] E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 228–266. Springer-Verlag, 1995.
- [8] J. Meseguer, editor. *Proceedings of the 1st International Workshop on Rewriting Logic and its Applications, RWLW'96, (Asilomar, Pacific Grove, CA, USA)*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Sept. 1996.
- [9] E. Visser. Language independent traversals for program transformation. In Jeuring [3], pages 86–104.
- [10] E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *International Conference on Functional Programming (ICFP'98), Baltimore, Maryland. ACM SIGPLAN*, pages 13–26, Sept. 1998.
- [11] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, Jan. 1992.