Strategic Pattern Matching

Eelco Visser*

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

Abstract. Stratego is a language for the specification of transformation rules and strategies for applying them. The basic actions of transformations are matching and building instantiations of first-order term patterns. The language supports concise formulation of generic and data type-specific term traversals. One of the unusual features of Stratego is the separation of scope from matching, allowing sharing of variables through traversals. The combination of first-order patterns with strategies forms an expressive formalism for pattern matching. In this paper we discuss three examples of *strategic pattern matching*: (1) *Contextual rules* allow matching and replacement of a pattern at an arbitrary depth of a subterm of the root pattern. (2) *Recursive patterns* can be used to characterize concisely the structure of languages that form a restriction of a larger language. (3) *Overlays* serve to hide the representation of a language in another (more generic) language. These techniques are illustrated by means of specifications in Stratego.

1 Introduction

First-order terms are used to represent data structures in term rewriting systems, functional and logic programming languages. First-order patterns are used to decompose such terms by simultaneously recognizing a structure and binding variables to subterms, which would otherwise be expressed by nested conditional expressions that test tags and select subterms. However, first-order patterns are not treated as first-class citizens and their use poses limitations on modularity and reuse: no abstraction over patterns is provided because they may occur only in the left-hand side of a rewrite rule, the arms of a case, or the heads of clauses; pattern matching is at odds with abstract data types because it exposes the data representation; a first-order pattern can only span a fixed distance from the root of the pattern to its leafs, which makes it necessary to define recursive traversals of a data structure separately from the pattern to get all needed information.

For these reasons, enhancements of the basic pattern matching features have been implemented or considered for several languages. For example, *list matching*

* This paper was written while the author was employed by the Pacific Software Research Center, Oregon Graduate Institute, Portland, Oregon, USA. This work was supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069.

P. Narendran and M. Rusinowitch (Eds.): RTA'99, LNCS 1631, pp. 30–44, 1999.

in ASF+SDF [7] is used to divide a list in multiple sublists possibly separated by element patterns. *Associative-commutative (AC) matching* in OBJ, Maude [5] and ELAN [3] supports the treatment of lists as multi-sets. *Higher-order matching* in λProlog [17] allows the matching of subterms at a variable depth. *Views* for Haskell, as proposed in [24], provide a way to view a data structure using different patterns than are used to represent them. Each of these techniques provides a mix of structure recognition, variable binding, *term traversal*, and *transformation*. For instance, in list (AC, higher-order) matching a term is first transformed by application of the associative and identity laws (associative and commutative laws, $\beta\eta$-conversion) and then matched against the given pattern. Matching a view pattern involves the transformation of the underlying data structure to the view data type.

This paper shows how the *rewriting strategies* paradigm [3,5,14,22,23] provides a general framework for describing and implementing such pattern matching combinations. Rewriting strategies are programs that determine the order in which rewriting rules are applied. One of the important aspects of strategies is the definition of term traversals to find subterms to which rules can be applied. Here the application of such traversals in the definition of patterns *at the level of individual rules* is considered.

This paper explores *strategic pattern matching* in Stratego [22,23], a language for the specification of program transformation systems. Stratego is a layer of syntactic abstractions on top of System S, a core language for the definition of rewriting strategies. The basic actions of System S are matching and building instantiations of first-order term patterns. The language supports concise formulation of generic and data type specific term traversals. One of the unusual features of System S is the separation of scope from matching, allowing sharing of variables through traversals. The combination of first-order patterns with strategies forms an expressive formalism for pattern matching.

The next section gives a brief overview of System S and Stratego. The following sections discuss three applications of strategic pattern matching illustrated by means of specifications in Stratego: (1) Contextual rules allow matching and replacement of a subterm at an arbitrary depth with respect to the root of a pattern. Section 3 shows how contextual patterns are used in a concise specification of a type checker. (2) Recursive patterns can be used as predicates to characterize concisely the structure of languages that form a subset of a larger language. Section 4 illustrates the idea by means of a characterization of conjunctive and disjunctive normal forms as a restriction of propositional formulae. Section 5 applies the same technique to characterize the embedding of AsFix, the abstract syntax representation of ASF+SDF, into ATerms, a universal data type. (3) Overlays are pseudo-constructors that abstract from an underlying (complex) representation using real constructors. They can be used to overlay a language on top of another more generic representation language. Section 5 defines overlays for AsFix to hide the details of its embedding in ATerms. Related work is discussed in Section 6 and some conclusions are drawn in Section 7.

2 Rewriting Strategies

This section introduces System S, a calculus for the definition of tree transformations, and Stratego, a specification language providing syntactic abstractions for System S expressions. For an operational semantics see [22,23].

2.1 System S

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

First-order terms are expressions over the grammar

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

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

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

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

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

Fig. 1. Specification of several generic term traversal strategies.

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

Figure 1 defines various traversals based on these operators. For instance, `oncetd(s)` tries to find *one* application of `s` somewhere in the term starting at the root working its way down; `s <+ one(x)` first attempts to apply `s`, if that fails an application of `s` is (recursively) attempted at one of the children of the subject term. If no application is found the traversal fails. Compare this to the traversal `alltd(s)`, which finds *all* outermost applications of `s` and never fails.

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

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

A build `!t` replaces the subject term with the instantiation of the pattern `t` using the current bindings of terms to variables in `t`. A scope `{x1,...,xn: s}` makes the variables `xi` local to the strategy `s`. This means that bindings to these variables outside the scope are undone when entering the scope and are restored

after leaving it. The operation `where(s)` applies the strategy `s` to the subject term. If successful, it restores the original subject term, keeping only the newly obtained bindings to variables.

2.2 Stratego

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

A signature declares the sorts and operations (constructors) that make up the structure of the language(s) being transformed. An example signature is shown in Figure 2. A strategy definition `f(x1,...,xn) = s` introduces a new strategy operator `f` parameterized with strategies `x1` through `xn` and with body `s`. Such definitions cannot be recursive, i.e., they cannot refer (directly or indirectly) to the operator being defined. All recursion must be expressed explicitly by means of the recursion operator `rec`. Labeled transformation rules are abbreviations of a particular form of strategy definitions. A conditional rule `L : l -> r where s` with label `L`, left-hand side `l`, right-hand side `r`, and condition `s` denotes a strategy definition `L = {x1,...,xn: ?l; where(s); !r}`. Here, the body of the rule first matches the left-hand side, and then attempts to satisfy the condition `s`. If that succeeds, then it builds the right-hand side `r`. The rule is enclosed in a scope that makes all term variables `xi` occurring in `l`, `s` and `r` local to the rule. If more than one definition is provided with the same name, e.g., `f(xs) = s1` and `f(xs) = s2`, this is equivalent to a single definition with the sum of the original bodies as body, i.e., `f(xs) = s1 + s2`.

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

This paper is about three programming idioms and the syntactic abstractions to support them. Recursive patterns are an idiom that is directly supported by Stratego as introduced above. The syntax of Stratego has been extended for contexts and overlays to provide more concise syntax for these idioms. However, these syntax extensions are implemented without extending System S.

2.3 Implementation

The Stratego compiler translates a specification to a C program that reads a term, applies the specified transformation to it, and, if succesful, outputs the transformed term. 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 [19]. The compiler is implemented in Stratego itself.

```
module pico-syntax
imports list-basic
signature
  sorts Program Decl Stat Expr Type Id
  operations
    Block   : List(Decl) * Stat -> Program
    Decl    : Id * Type  -> Decl
    Natural : Type                          Plus  : Expr * Expr -> Expr
    String  : Type                          Minus : Expr * Expr -> Expr
    Skip    : Stat                          Conc  : Expr * Expr -> Expr
    Assign  : Id * Expr -> Stat             Var   : Id -> Expr
    Seq     : Stat * Stat -> Stat           Int   : Int -> Expr
    If      : Expr * Stat * Stat -> Stat    Str   : String -> Expr
    While   : Expr * Stat -> Stat           Id    : String -> Id
```

Fig. 2. Abstract syntax of Pico.

3 Contexts

This section describes *contextual patterns*, i.e., patterns that relate some bit of information from the root pattern to a subterm at variable depth. This is illustrated by the specification of a type checker for the toy language Pico [7]. Heering [11] gives a concise specification of such a typechecker using a combination of second-order matching to relate variable declarations and their occurrences in a program and an abstract interpretation style of type checking.

Pico is a small imperative while-language. It has expressions ranging over natural number and string values and the usual statement combinators. A program consists of a block, which contains a list of variable declarations and a statement. Variable declarations associate a type (`Natural` or `String`) with a variable identifier. The abstract syntax of Pico is defined in Figure 2.

A program is statically correct if variables are used consistently with their declarations. Conventionally, type checkers are defined as a predicate that traverses the program carrying the declarations and checking the correctness of expressions and statements. In the abstract interpretation style of [11] first all variable occurrences are replaced by their types (using an injection `Tp` of types into identifiers), then consistent combinations of such typed expressions and statements are reduced to simpler forms. For example, let variables `"a"` and `"b"` have type `String`, the expression `Conc(Var(Id("a")),Var(Id("b")))` is first transformed into `Conc(Var(Tp(String)),Var(Tp(String)))`, which then reduces to `Var(Tp(String))`. If the program is correct it will reduce to a block with a skip statement. However, if the program contains type errors, residuals of this error will remain in the result of the type checking procedure and point to the offending parts of the program. For example, the program

```
Block([Decl(Id("a"), Natural), Decl(Id("b"), String)],
While(Var(Id("a")), Assign(Id("b"), Plus(Var(Id("a")), Var(Id("a"))))))
```

reduces to

```
module pico-typecheck
imports pico-syntax traversals
signature
  operations
    Tp : Type -> Id
rules
  InlTp : Block(ds[Decl(Id(x), t)], s[Id(x)]) -> Block(ds, s[Tp(t)])
  IntTp : Int(n) -> Var(Tp(Natural))
  StrTp : Str(s) -> Var(Tp(String))
  Check : Seq(Skip, s) -> s
  Check : Seq(s, Skip) -> s
  Check : Assign(Tp(t), Var(Tp(t))) -> Skip
  Check : If(Var(Tp(Natural)), s1, s2) -> Seq(s2, s3)
  Check : While(Var(Tp(Natural)), s1) -> s1
  Check : Plus(Var(Tp(Natural)), Var(Tp(Natural))) -> Var(Tp(Natural))
  Check : Minus(Var(Tp(Natural)), Var(Tp(Natural))) -> Var(Tp(Natural))
  Check : Conc(Var(Tp(String)), Var(Tp(String))) -> Var(Tp(String))
strategies
  typecheck = downup2(repeat(InlTp + IntTp + StrTp), repeat(Check))
```

Fig. 3. Type checking rules and strategy for Pico.

```
Block([Decl(Id("a"),Natural), Decl(Id("b"), String)],
 Assign(Tp(String), Var(Tp(Natural))))
```

making clear that the assignment statement is not type correct.

A specification of this approach is shown in Figure 3. The `typecheck` strategy declares a `downup2` traversal over the program. On the way down identifiers and constants are replaced by their types by means of rules `InlTp`, `IntTp` and `StrTp`. On the way up well-typed expressions and statements are reduced to simpler forms by the `Check` rules. Distribution of type information over a program is achieved by means of the contextual rule `InlTp`. The sub-pattern `ds[Decl(Id(x), t)]` is a context that matches one instance of the pattern `Decl(Id(x), t)` as a subterm of the `ds` argument of the `Block` pattern. The sub-patterns `s[Id(x)]` in the left-hand side and `s[Tp(t)]` in the right-hand side form a context that replaces one occurrence of `Id(x)` somewhere in the statements by `Tp(t)`, where `x` and `t` are determined by the match in the `ds` context.

Contextual rules are implemented by translation to primitive constructs. A context `x[t]`, occurring on the left-hand side only, corresponds to a traversal over the term matching `x` trying to find a match of the pattern `t`. A context `x[l]` in the left-hand side and `x[r]` in the right-hand side corresponds to a traversal over the term matching `x` that replaces an occurrence of `l` by the corresponding instantiation of `r`. Thus, a first attempt at implementation of rule `InlTp` is:

```
InlTp : Block(ds, s) -> Block(ds, s')
        where <oncetd(?Decl(Id(x), t))> ds;
              <oncetd(?Id(x); !Tp(t))> s => s'
```

The first clause in the condition makes a traversal over the declarations finding a declaration. The second clause traverses the statements replacing an occurrence of the identifier in the declaration by its type.

However, this does not achieve the desired effect. If the first traversal finds a declaration for which there are no (more) occurrences of the identifier in the statements, then the second traversal will fail, even if there are other declarations for which it would succeed. In other words the first traversal needs to backtrack to find other declarations if the second traversal fails. This is achieved by inlining the second traversal in the first, as follows:

```
InlTp : Block(ds, s) -> Block(ds, s')
        where <oncetd(?Decl(Id(x), t);
                      where(<oncetd(?Id(x); !Tp(t))> s => s'))> ds
```

The `where` clause of this rule computes a new value `s'`. The outer traversal walks over the declarations. When a declaration for an identifier `Id(x)` is found the inner traversal walks over the statements and replaces one occurrence of `Id(x)` with its type `Tp(t)` from the declaration. If no occurrence of the identifier is found in the statements the outer loop continues to search for another declaration. If no declaration and matching identifier occurrence in the statements can be found, the rule fails.

4 Recursive Patterns

This section treats *recursive patterns*, i.e., patterns that describe recursive structure as opposed to the fixed structure described by first-order patterns. The idiom of recursive patterns is illustrated by the specification of language restrictions. Recursive patterns are also useful tools in program analysis.

A signature generates a language of terms. A language restriction is a subset of a language. Restrictions are not always syntactic, i.e., do not correspond to the language generated by a subsignature, but can require constructs only to be used in certain combinations. Examples of language restrictions abound in language processing: (1) The set of normal forms with respect to a set of rewrite rules is a restriction. The rewrite rules give an operational method for obtaining the normal form of a term, but they do not describe the *structure* of the normal forms. (2) A core language reflects the computational kernel of a language. Again, the transformation that translates a program in the complete language to a core language program does not define the structure of core language programs. (3) The intermediate languages produced by the stages of a compiler are often restrictions of a common language. Subsequent stages introduce lower-level features. The combination of all constructs might not form a valid language. (4) Languages embedded in a generic representation format. The generic format allows a wide range of expressions, only a few of those are expressions in the embedded language.

Language restrictions are often dealt with informally. A component of a language processor assumes its input to be in a certain form that is not defined anywhere. Descriptions of language restrictions separate from the transformations

```
module prop
signature
  sorts Prop
  operations
    Atom : String -> Prop    And : Prop * Prop -> Prop
    Not  : Prop -> Prop      Or  : Prop * Prop -> Prop
strategies
  conj(s) = rec x(And(x, x) <+ s)
  disj(s) = rec x(Or(x, x)  <+ s)
  conj-nf = conj(disj(Atom(id) + Not(Atom(id))))
  disj-nf = disj(conj(Atom(id) + Not(Atom(id))))
```

Fig. 4. Characterization of conjunctive and disjunctive normal forms.

that produce them are useful for documentation (what restriction is consumed or produced by this language processor) and validation (check that the input or output of a processor conforms to the restriction). Strategies support the concise description of language restrictions by means of *recursive patterns*. A recursive pattern is a strategy that describes the structure of a set of terms by means of recursion and congruences. This technique is illustrated by two examples: disjunctive and conjunctive normal forms of propositional formulae and, in the next section, the embedding of AsFix in ATerms.

As a first example, consider a language of propositional formulae constructed from atoms (proposition letters) with negation, conjunction and disjunction. The signature describing the abstract syntax of this language is shown in Figure 4.

A formula is in conjunctive normal form if it is a conjunction of disjunctions of atoms or negated atoms. Likewise, a formula is in disjunctive normal form if it is a disjunction of conjunctions of atoms or negated atoms. These restrictions can be characterized concisely by means of the recursive patterns in Figure 4. Given some strategy `s` that characterizes formulae in some form, the strategy `rec x(And(x, x) <+ s)` describes conjunctions of the form `And(And(..., ...), And(..., ...))` with leaves of the form `s`. Thus, the operators `conj(s)` and `disj(s)`, describe conjuncts and disjuncts of `s`'s, respectively. Hence, the combination `conj(disj(s))` describes conjuncts of disjuncts of `s`'s. Unfolding the definition of `conj` and `disj` in `conj-nf` gives:

```
conj-nf = rec x(And(x, x) + rec y(Or(y, y) + Not(Atom(id)) + Atom(id)))
```

So `conj-nf` and `disj-nf` describe conjunctive and disjunctive normal forms, respectively.

5 Overlays

This section introduces *overlay patterns*, i.e., patterns composed with pseudo-constructors that abstract from a concrete representation with real constructors. Overlays are first-class citizens in the sense that all operations that apply to

normal constructors, i.e., matching, building and congruence, apply to overlays as well. Furthermore, overlays can be defined in terms of other overlays, allowing a hierarchy of abstractions. The technique is illustrated by the definition of overlays for the representation of AsFix constructs in ATerms. These are applied in another example of recursive patterns to describe the restriction of ATerms to AsFix expressions.

5.1 ATerms and AsFix

The Annotated Term Format or ATerms [19] is a universal data type designed for representing data types in a generic manner for the purpose of data exchange, generic manipulation and persistent storage of data. Figure 5 gives the signature of ATerms. An ATerm is either an application (`Appl`) of an `AFun` to a list of terms, or a list (`AList`) of terms. An `AFun` is either an integer (`Int`), quoted string (`Str`) or an unquoted symbol (`Sym`). For example, the ATerm

```
Appl(Sym("And"), [Appl(Sym("Atom"), [Appl(Str("a"), [])]),
                  Appl(Sym("Atom"), [Appl(Str("b"), [])])])
```

is an encoding of the propositional formula `And(Atom("a"), Atom("b"))`.

AsFix is the abstract syntax for the algebraic specification formalism ASF+SDF [7]. It is used as the intermediate representation for language processors such as a term rewriting compiler and a pretty-printer generator. The AsFix representation of a specification consists of a signature and a list of conditional equations over typed first-order terms. Here only unconditional equations over first-order terms are considered.

One of the characteristics of AsFix is its encoding of syntactic information. In ASF+SDF constructors are defined by means of a context-free production that declares its mix-fix syntax and the sorts of its arguments. In AsFix both the sort information and the syntactic information is retained. For example, the production `E "+" E -> E` is represented by the AsFix expression

```
Prod([Sort("E"), Lit("+"), Sort("E")], Sort("E"))
```

Productions `p` of this form are used as the constructor 'names' in applications of the form `App(p, [a1,...,an])`. Below a precise definition of AsFix is given.

AsFix expressions can be represented as ATerms. For instance, the production above is represented by the ATerm

```
Appl(Sym("Prod"), [AList([Appl(Sym("Sort"), [Appl(Str("E"), [])]),
                          Appl(Sym("Lit"),  [Appl(Str("+"), [])]),
                          Appl(Sym("Sort"), [Appl(Str("E"), [])])]),
                   Appl(Sym("Sort"), [Appl(Str("E"), [])])])
```

This representation allows easy exchange, persistency and generic manipulation of AsFix expressions. However, the representation has two problems: (1) Since the ATerm format is a universal datatype, not every ATerm is a valid AsFix expression. (2) Since the ATerm format is bulky, specifying operations on AsFix using pattern matching on the ATerm representation is rather tedious. The first

```
module aterms
signature
  sorts AFun ATerm
  operations
    Int   : Int -> AFun        Appl  : AFun * List(ATerm) -> ATerm
    Str   : String -> AFun     AList : List(ATerm) -> ATerm
    Sym   : String -> AFun
```

Fig. 5. ATerm signature.

problem is solved by defininining a recursive pattern that characterizes the ATerms that are valid AsFix expressions. This recursive pattern can be used to validate input to language processors. The second problem is solved by defining overlays that abstract from the concrete ATerm representation of AsFix expressions, while still maintaining that representation under the hood.

5.2 Overlays for AsFix

Overlays are abstractions of term patterns. An overlay definition `C(x1, ..., xn) = pat` introduces a new constructor `C` with `n` arguments that is an abbreviation of the pattern `pat`. This new constructor can be used in all places where the pattern `pat` can be used, i.e., in match patterns, build patterns and congruences. An expression `?t` (`!t`) with an occurrence of `C(t1,...,tn)` denotes the expression `?t'` (`!t'`), where `t'` is obtained by replacing `C(t1,...,tn)` by `pat[t1/x1,...,tn/xn]` in `t`. A congruence expression `C(s1,...,sn)` denotes the instantiation of the congruence derived from the pattern `t`, with the strategies `si` substituted for the variables `xi`.

Figure 6 defines overlays for the constructs of AsFix. For example, the overlay

```
Prod(as, r) = Appl(Sym("Prod"), [AList(as), r])
```

defines an abstraction for the ATerm pattern encoding an AsFix production. Using these overlays the complicated ATerm above can be written as

```
Prod([Sort("E"), Lit("+"), Sort("E")], Sort("E"))
```

Overlays can now be used in the recursive pattern that characterizes the restriction of ATerms to AsFix. The patterns `asfix-...` in Figure 6 describe the syntactic categories of AsFix expresions using the congruences — expressions such as `Lit(string)` and `App(asfix-prod, list(x))` — that are derived from the overlays. The pattern `asfix-prod` defines a production as a `Prod` with a list of `asfix-sorts` as first argument and an `asfix-sort` as second argument. The pattern `asfix-term` defines an AsFix term as a literal, a typed variable, or an application of a production to a list of terms. The pattern `asfix-equ` defines an equation as an `Equation` with a non-variable term as left-hand side and a term as right-hand side.

The recursive patterns only describes 'raw' AsFix expressions and do not check that the argument sorts in the `Prod` of an application correspond to the

```
module asfix
imports aterms traversals strings
overlays
  Sort(n)        = Appl(Sym("Sort"), [Appl(Str(n), [])])
  Lit(l)         = Appl(Sym("Lit"), [Appl(Str(l), [])])
  Prod(as, r)    = Appl(Sym("Prod"), [AList(as), r])
  App(p, as)     = Appl(Sym("App"), [p, AList(as)])
  Var(s, n)      = Appl(Sym("Var"), [s, n])
  Equation(l, r) = Appl(Sym("Equation"), [l, r])
strategies
  asfix-sort = Sort(string) + Lit(string)
  asfix-prod = Prod(list(asfix-sort), asfix-sort)
  asfix-term = rec x(Lit(string) +
                     Var(asfix-sort, string) +
                     App(asfix-prod, list(x)))
  asfix-equ  = Equation(asfix-term; not(Var(id, id)), asfix-term)
  asfix-eqs  = list(asfix-equ)
rules
  Check1 : Var(x, _) -> x
  Check2 : App(Prod(args, res), args) -> res
  Check3 : Equation(srt, srt) -> Equation(srt, srt)
strategies
  typecheck-term = rec x(Lit(id) + Check1 + App(id, list(x)); Check2)
  typecheck-eqs = list(Equation(typecheck-term, typecheck-term); Check2)
```

Fig. 6. AsFix: overlays, recursive pattern and type checker.

sorts of the actual arguments of the application. To test this a typechecker in an abstract interpretation style similar to that of the Pico typechecker in Section 3 is defined in Figure 6. Only now there is no need to distribute type information, since terms are already annotated with their types.

The process of defining overlays to hide the underlying representation can be repeated, e.g., to define on top of the AsFix abstractions another layer to describe patterns for a specific instantiation of AsFix terms. For instance, take the SDF productions `E "+" E -> E` and `E "*" E -> E`. The following overlays define shorthands for AsFix terms using these productions:

```
overlays
  BinexpOp(o)     = Prod([Sort("E"), Lit(o), Sort("E")], Sort("E"))
  Binexp(l, o, r) = App(Binexp(o), [l, Lit(o), r])
  Plus(l, r)      = Binexp(l, "+", r)
  Mul(l, r)       = Binexp(l, "*", r)
```

These overlays allow the 'domain-specific' transformation rule

```
Distr : Mul(x, Plus(y, z)) -> Plus(Mul(x, y), Mul(x, z))
```

Although this rule is written at the level of the embedded language of expressions, they are applied at the level of the underlying ATerm representation.

6 Related Work

Programmable rewriting strategies originate in theorem proving tactics and were first introduced in rewriting in the specification language ELAN [13]. In the algebraic specification formalism Maude [5] strategies can be defined by the user as meta-level specifications. System S and Stratego were developed in [14] (sequential non-deterministic programming and generic term traversal) and [22,23] (breaking down rewrite rules in matching and building term patterns). ELAN supports congruences and recursive equations, which should support definition of recursive patterns. Overlays and contexts are not supported either by ELAN or by Maude. See [22,23] for more details on the relation between these languages.

A wide range of languages introduce enhanced pattern matching features. A brief and necessarily incomplete overview follows:

The transformation languages Dora [10] and TXL [6,15] are examples of languages with some ad-hoc combinations of traversal and pattern matching.

Context patterns can be implemented by means of higher-order matching in λProlog [17]. A higher-order pattern `F(t)` instantiates the function variable `F` such that application to `t` yields the term that is matched. Heering [11] gives an example of second-order matching that we discussed in Section 3. Mohnen's context patterns for Haskell [16] are similar to higher-order matching in λProlog. Sellink and Verhoef [20] show how list matching can be used to implement shallow contexts (that can find statements in a list of statements, but at a fixed nesting depth) used for transforming COBOL programs. Stratego contexts provide the additional possibility of specifying the traversal to be used. This implies that restrictions on the structure of the context can be imposed and that more than one replacement can be done.

Aiken and Murphy [1] describe a language of regular tree expressions for program analysis. Their language is very similar to the recursive patterns in this paper, but is restricted to recognition only.

An overlay is an abbreviation for three abstractions: a match abstraction, a build abstraction and a congruence abstraction. The pattern templates for SML of Aitken and Reppy [2] define two abstractions: a match abstraction and a build abstraction. Congruences are not supported in SML. Another difference with templates is that templates need to be linear. In a definition `C(x) = t`, the variable `x` can occur only once in `t`. Overlays do not need to be linear.

A view type in Wadler's proposal for views in Haskell [24] presents an alternative view to a representation data type by means of a pair of conversion functions that translate between the representation type and the view type. Views are more general than overlays and templates, in that they allow rearrangement of the underlying pattern. However, this added expressivity turns into a disadvantage if one considers pattern matching. Overlays are abstractions that do not result in a loss of efficiency, while views can require an arbitrary transformation. View-like transformations are of course expressible in Stratego. Thompson's lawful types [21] for Miranda are similar to views.

Another problem with general views is that it can destroy equational reasoning [4]. Several proposals [4,8,9,18] repair this by only allowing views in match

expressions and not in build expressions. Values in the underlying data representation should be constructed by means of functions. Erwig's active patterns [8] could be considered as functions that inspect and transform their argument and then bind subterms to variables or fail. Fähndrich and Boyland [9] syntactically restrict the patterns used in pattern abstractions such that pattern matching becomes statically checkable.

7 Conclusions

This paper presented three examples of strategic pattern matching: contexts, recursive patterns and overlays. These idioms provide concise specification of expressive patterns that enhance standard first-order patterns. Their definition follows naturally from the features of System S; for some of the techniques new syntactic abstractions were added to Stratego, but no new System S constructs were needed. The key features that enable this expressivity are: (1) ability to abstract over pattern matching (where abstraction over building is a common feature of many languages), and (2) the separation of variable scope and matching, which enables the communication of variable bindings over other operations, and (3) generic term traversals through `all`, `some` and `one`.

The techniques described in this paper have been applied in the specification of the (bootstrapped) Stratego compiler, in an optimizer for RML [23], and in a specification of the warm fusion transformation for functional programs [12]. Future work includes: the application of these techniques in other program transformations; the development of more abstractions for concise specification of program transformations; and the optimization of strategies, in particular traversal fusion, which is important for the optimization of contextual rules.

Acknowledgements The author thanks Andrew Tolmach, Patty Johann and the referees for comments on drafts of this paper.

References

1. A. Aiken and B. Murphy. Implementing regular tree expressions. In *Functional Programming and Computer Architecture (FPCA'91)*, pages 427–447, Aug. 1991.
2. W. E. Aitken and J. H. Reppy. Abstract value constructors. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 1–11, San Francisco, Cal., June 1992.
3. P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar, Pacific Grove, CA, September 1996. Elsevier.
4. F. W. Burton and R. D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, April 1993.
5. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89, Asilomar, Pacific Grove, CA, September 1996. Elsevier.

6. J. R. Cordy, I. H. Carmichael, and R. Halliday. *The TXL Programming Language, Version 8*, Apr. 1995.
7. A. Van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, Singapore, September 1996.
8. M. Erwig. Active patterns. In *Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*, pages 21–40, 1996.
9. M. Fähndrich and J. Boyland. Statically checkable pattern abstractions. In *International Conference on Functional Programming (ICFP'97)*, pages 75–84, Amsterdam, The Netherlands, June 1997. ACM SIGPLAN.
10. C. D. Farnum. *Pattern-Based Languages for Prototyping of Compiler Optimizers*. PhD thesis, University of California, Berkeley, 1990. Technical Report CSD-90-608.
11. J. Heering. Second-order term rewriting specification of static semantics: An exercise. In Van Deursen et al. [7], chapter 8, pages 295–305.
12. P. Johann and E. Visser. Warm fusion in Stratego. A case study in generation of program transformation systems. Technical report, Department of Computer Science, Universiteit Utrecht, 1999. http://www.cs.uu.nl/∼visser/stratego/.
13. C. Kirchner, H. Kirchner, and M. Vittek. Implementing computational systems with constraints. In P. Kanellakis, J.-L. Lassez, and V. Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming*, pages 166–175, Providence R.I., USA, 1993. Brown University.
14. B. Luttik and E. 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.
15. A. Malton. The denotational semantics of a functional tree-manipulation language. *Computer Languages*, 19(3):157–168, 1993.
16. M. Mohnen. Context patterns, part ii. In *Implementation of Functional Languages*, pages 338–357, 1997.
17. G. Nadathur and D. Miller. An overview of λProlog. In R. A. Kowalski, editor, *Logic Programming. Proceedings of the Fifth International Conference and Symposium*, volume 1, pages 810–827, Cambridge, Mass., USA, 1988. MIT Press.
18. C. Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML*, pages 14–23, Baltimore, Maryland, USA, September 1998.
19. P. A. Olivier and H. A. de Jong. Efficient annotated terms. Technical report, Programming Research Group, University of Amsterdam, August 1998.
20. M. P. A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the 5-th Working Conference on Reverse Engineering (WCRE'98)*, pages 89–103, Honolulu, Hawaii, USA, October 1998.
21. S. Thompson. Laws in Miranda. In *ACM Symposium on Lisp and Functional Programming*, pages 1–12. ACM, August 1986.
22. E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, Electronic Notes in Theoretical Computer Science, Pont-à-Mousson, France, September 1–4 1998. Elsevier.
23. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming (ICFP'98)*, pages 13–26, Baltimore, Maryland, September 1998. ACM.
24. P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 307–313, Munich, January 1987. ACM.