Electronic Notes in Theoretical Computer Science 58 No. 2 (2001)
URL: `http://www.elsevier.nl/locate/entcs/volume58.html` 17 pages

A Logic for Rewriting Strategies

Richard B. Kieburtz [1], [2]

Oregon Graduate Institute
20000 N.W. Walker Road
Beaverton, OR 97006 USA

Abstract

Rewriting strategies can become quite complex and are not easy to comprehend or reason about when they are expressed in operational terms. This paper develops a *weakest precondition* logic for reasoning about strategies programmed in the strategy language *Stratego*. This logic embeds the modal mu-calculus, allowing it to express properties of terms of arbitrary depth. Its use is illustrated by characterizing properties of several reduction strategies for the lambda calculus with explicit substitutions.

1 Introduction

Strategies for term rewriting are widely used to implement syntactic theories in systems for automated deduction, including theorem-proving and program transformation. Strategies evaluate conditions for the application of rewrite rules, determine the order in which subterms are explored, and prescribe bindings and scope of pattern variables. Formulating strategies is a programming task that can be as complex as any other that we know. It can be made easier with appropriate programming language support and better understood through logical characterization.

This paper is a first step towards defining a programming logic for strategies. Stragegies are understood as programs over a domain of terms. Control of strategies is accomplished with recursion and nondeterministic choice. A *weakest-precondition* logic furnishes a natural formalism for reasoning about such programs. However, predicates in this logic are interpreted over a domain of term structures. As a logic for terms, we have adopted the μ-calculus [7], enriched with modalities that express path quantification in terms.

[1] Email: `dickcse.ogi.edu`

[2] This research was done while the author was a Visiting Professor at INRIA-Rocquencourt laboratory. Partial support was provided by the National Science Foundation under Grant No. CCS-9803880.

Rules have been developed in this logic for the constructions of *Stratego* [12,13,15,14], a domain-specific language designed specifically for programming strategies. *Stratego* provides a compositional semantics with explicit recursion, allowing strategies to be applied at sites deep within terms. Strategies, and therefore patterns, are first-class constructs of *Stratego*. As in a logic programming language, conditional control in *Stratego* is based upon the success or finite failure of strategies, rather than if-then-else expressions that test conditions coded as boolean values. Although essentially a first-order language, *Stratego* also supports a particular form of higher-order strategies, namely *term congruences*, which lift term constructors into strategy constructors.

Section 2 introduces a weakest-precondition (*wp*) logic for strategies. In Section 3, several strategies for reducing terms in the lambda-calculus are proposed and properties of reduction strategies are characterized in the *wp* logic. Section 4 discusses related work and Section 5 presents conclusions.

2 A weakest-precondition logic for strategies

Rewriting strategies are designed to produce terms that exhibit particular forms, by a directed series of rewriting steps. To reason about strategic rewriting, we'd like to know a set of input terms from which a given rewriting strategy is assured to produce an output term in a specified form. A predicate characterizing the largest set of such input terms is a *weakest precondition* for the strategy to produce a specified form of output.

In a weakest-precondition logic, each rewrite rule or strategy is characterized by a *predicate transformer*, a function from predicates to predicates. Since predicates characterize sets in a given universe, we can think of a *wp*-logic as interpreted in relations over a universe.

The universe we have in mind is a Herbrand universe of terms generated by a finite signature, Σ. Call this universe $\mathcal{T}(\Sigma)$. Predicates are interpreted as subsets of this universe, by an interpretation function, $\mathcal{I} : \mathit{Pred} \to \mathcal{T}(\Sigma)$. The distinguished predicates *True* and *False* have the interpretations $\mathcal{I}(\mathit{True}) = \mathcal{T}(\Sigma)$ and $\mathcal{I}(\mathit{False}) = \emptyset$. Term variables in the logic range over the universe.

Connectives $(\neg)$, $(\vee)$, $(\wedge)$ and $(\Rightarrow)$ are used to form compound predicate formulas. They have the interpretations

$$\begin{aligned} \mathcal{I}(\neg P) &= \{t \mid t \notin \mathcal{I}(P)\} \\ \mathcal{I}(P \vee Q) &= \mathcal{I}(P) \cup \mathcal{I}(Q) \\ \mathcal{I}(P \wedge Q) &= \mathcal{I}(P) \cap \mathcal{I}(Q) \\ \mathcal{I}(P \Rightarrow Q) &= \mathcal{I}(P) \subseteq \mathcal{I}(Q) \end{aligned}$$

In the sequel, we shall indulge in a common abuse of notation by using predicates to denote the sets that they characterize.

If s is a strategy for term rewriting, then by $wp_s : \mathit{Pred} \to \mathit{Pred}$ we denote the predicate transformer associated to s. The expression $\langle s \rangle\ t$ denotes the application of strategy s to term t. We denote by $Dom(s)$ the set $\{t \mid \langle s\rangle\, t \in \mathcal{I}(True)\}$. That is, $Dom(s)$ is the set on which strategy s succeeds.

We also require a predicate characterizing terms on which a strategy, s, fails. This requires a bit more subtlety than just taking the complement of $Dom(s)$, because finite failure of a strategy is used for control, whereas failure by nontermination obviously cannot be. Denote by $\overline{Dom(s)}$ the set on which strategy s fails finitely. In case strategy s terminates uniformly, $Dom(s) \vee \overline{Dom(s)} = True$.

2.1 Rules of **wp** logic for Stratego

The weakest-precondition logic satisfies several rules, including

$$\frac{P \Rightarrow Q}{wp_s(P) \Rightarrow wp_s(Q)}$$

$$wp_s(True) = Dom(s) \qquad wp_s(False) = False$$

$$wp_{s;t}(P) = wp_s(wp_t(P))$$

$$wp_{s<+t}(P) = wp_s(P) \;\vee\; (wp_t(P) \wedge \overline{Dom(s)})$$

$$wp_{s+t}(P) = (wp_s(P) \wedge \overline{Dom(t)}) \;\vee\; (wp_t(P) \wedge \overline{Dom(s)})$$

$$wp_s(\neg P) \wedge wp_s(P) = False$$

The rules for alternatives combinators $(+)$ and $(<+)$ are those of *committed choice*. With committed choice, a potential choice strategy will not be effective on a term if there is an alternative choice that might either have succeeded or failed to terminate when applied to the current term.

A weakest precondition is intended to characterize the largest set from which a given strategy can be assured to produce a term satisfying a specified postcondition. However, this is a *constructive* interpretation. If $t \in wp_s(P)$, then the strategy application $\langle s\rangle t$ is accepted as constructive evidence of a term that satisfies P. Thus it should come as no surprise that the characterization it gives for nondeterministic choice $(+)$ is rather weak. For example,

$$\begin{aligned} wp_{s+s}(P) &= (wp_s(P) \wedge \overline{Dom(s)}) \vee (wp_s(P) \wedge \overline{Dom(s)}) \\ &= wp_s(P) \wedge \overline{Dom(s)} \\ &= False \end{aligned}$$

Since the strategy $s + s$ specifies a nondeterministic choice between two strategies that have exactly the same domain, there is no way to determine which of the two strategies is applied to produce a result. Thus interpreting an application of either strategy as evidence of a result would be constructively unsound.

We call this constructive interpretation the *weakest discriminating precondition*, because it specifies a domain on which the component strategies of a choice can be discriminated. The weakest discriminating precondition $wp_{r+s}(P)$ can supply definite information only over a domain on which strategies r and s both terminate but cannot both succeed.

Although it is tempting to give a more optimistic interpretation of nondeterministic choice, such an interpretation could give rise to difficulties. In a so-called *angelic* interpretation of nondeterminism, $wp_{s+s}(P) = wp_s(P)$, although that it cannot be said which of the possible choices succeeded in producing a result. This is incompatible with a committed choice semantics.

In the sequel, we shall only consider weakest discriminating preconditions.

2.2 Variables and environments

Thus far, we have discussed strategies as if only ground terms were transformed. However, the real power of rewriting strategies is only realized when we consider terms with variables. Term variables range over the ground terms in a universe. A term with variables may be valued as a ground term by providing an environment in which its variables are bound. A strategy can have the effect of binding variables in an environment, as well as transforming the term to which it is applied into a new form.

In *Stratego*, a binding environment for terms is implicit in every strategy. To express a property of a term, we shall need to express some properties of the environment.

Formally, an environment is a list of binding pairs, $[(x_1, t_1), (x_2, t_2), \ldots, (x_n, t_n)]$, in which $x_1, x_2, \ldots, x_n$ designate variables and $t_1, t_2, \ldots, t_n$ are ground terms. We shall write $[(x, t) \mid e]$ to designate an environment in which the pair (x, t) occurs at the head of the list. Environments are represented as lists rather than sets because a variable binding may be shadowed by the addition of new bindings of the same variable.

The fundamental judgment form is term equality, relative to an environment. Axioms of the judgment form include the usual axioms of equality, plus

$$[(x,t) \mid e] \models x = t \;:\; \mathit{True}$$

$$\frac{e \models x = t \;:\; \mathit{True}}{[(y,t') \mid e] \models x = t \;:\; \mathit{True} \quad (y \neq x)}$$

We say that an environment e_1 *refines* an environment e_2, writing $e_1 \prec e_2$, if e_1 is compatible with e_2 on all bindings visible in e_2, but it may also contain additional bindings.

$$e_1 \prec e_2 =_{def} \forall x, t.\, e_2 \models x = t \;:\; \mathit{True} \;\Rightarrow\; e_1 \models x = t \;:\; \mathit{True}$$

A weakest precondition asserts a predicate characterizing a set of (term, environment) pairs. It asserts properties of terms containing variables in a context in which the variables are bound.

Properties of elementary strategies will be characterized in terms of a small-step semantics. In the formulation below, $\Xi_{t,e}$ denotes the characteristic predicate of a set comprised of a single pair, $\{(t,e)\}$. The weakest precondition for a predicate P to hold of the result of a strategy, s is defined in terms of a predicate transformer, wp'_s, restricted to the transformation of singleton predicates. This elementary predicate transformer can be defined directly with term equality judgments for each primitive strategy.

This formulation allows the question of admissibility of a predicate to be considered separately from the formulation of *wp*-rules. We say that a predicate, P, is *admissible* if its satisfiability can be defined inductively from a basis of term-equality judgments.

$$Admissible(P) \Longleftrightarrow wp_s(P) = \{(t,e) \mid \exists t_1.\, (wp'_s(\Xi_{t_1,\emptyset}) \Rightarrow \Xi_{t,e}) \,\wedge\, \emptyset \models t_1 : P\}$$

The above definition allows us to calculate the weakest precondition of a composition of two strategies:

$$\begin{aligned} wp_{r;s}(P) &= wp_r(wp_s(P)) \\ &= wp_r(\mu\xi.\, \{(t,e) \mid \exists t_1.(wp'_s(\Xi_{t_1,\emptyset}) \Rightarrow \Xi_{t,e}) \wedge \\ &\qquad\qquad \emptyset \models t_1 : P \,\wedge\, e \models t : \xi\}) \\ &= \mu\xi.\, \{(t,e) \mid \exists t_1, t_2, e_2.\, (wp'_r(\Xi_{t_2,e_2}) \Rightarrow \Xi_{t,e}) \wedge \\ &\qquad\qquad (wp'_s(\Xi_{t_1,\emptyset}) \Rightarrow \Xi_{t_2,e_2}) \wedge \\ &\qquad\qquad \emptyset \models t_1 : P \,\wedge\, e \models t : \xi\} \end{aligned}$$

in which μ designates the least fixed-point binding operator of the μ-calculus [7].

This *wp* composition rule shows how a composition of strategies propagates refinements of an environment.

2.2.1 Pattern-matching and term-building strategies

Elementary strategies of *Stratego* include *pattern-matching*, which succeeds on terms that unify with the pattern given in the strategy, and *term-building*, which creates a new term, using a pattern given in the strategy and the bindings found in the current environment. Predicate transformations for the elementary strategies of pattern-matching and term building are:

$$wp_{?t}(P) = \{(t', e \mid \exists e'.\, e' \prec e \,\wedge\, e' \models t' = t : P\}$$

$$wp_{!t}(P) = \{(t', e) \mid e \models t : P\}$$

A pattern strategy succeeds if an initial term and the given pattern can be matched by instantiating variables. When it succeeds, it produces a refinement of the initial environment, binding variables that occur in the initial term or the pattern. The weakest precondition for a pattern strategy characterizes a set of initial term-environment pairs for which the term matches the pattern and satisfies the asserted postcondition.

The term-building rule does not introduce new bindings. It characterizes those environments in which the form given in the term-building strategy can satisfy the asserted postcondition. Since the result of a term-building strategy does not depend upon the initial term, the weakest precondition imposes no restriction on initial terms.

2.2.2 The test(s) strategy

The strategy *test*(s) succeeds whenever the strategy s succeeds, but it does not commit the bindings of variables made by s and restores the original term. The weakest precondition can be expressed as

$$wp_{test(s)}(P) = \{(t, e) \mid (t, e) \in wp_s(\mathit{True}) \wedge e \models t : P\}$$

2.2.3 Restricting the scope of variables

New variables may be introduced in the scope of a particular strategy. The notation $[(x, \perp) \mid e]$ indicates an environment in which x is a new (unbound) variable that shadows any previous binding for a variable of the same name. Then

$$wp_{\{x_1,\ldots,x_m:\,s\}}(P) = \{(t, e) \mid (t, [(x_1, \perp), \ldots, (x_m, \perp) \mid e]) \in wp_s(P)\}$$

2.2.4 Derived strategies

Many complex strategies can be defined in terms of these basic binding and building strategies. For instance, a strategy that applies a given strategy, s to a specific term is $!t;\ s$. This strategy can be written in a function-application style with the syntax $\langle s\rangle\ t$.

Another example is a *rule* in *Stratego*, which has the (sugared) syntax $\backslash\ p\ \rightarrow\ t$ **where** $r\ \backslash$. A rule is defined in terms of the compound strategy $\{x_1, \ldots, x_n\} : ?p;\ r;\ !t$, with the side condition that $FV(p) \cup FV(r) \cup FV(t) \subseteq \{x_1, \ldots, x_n\}$, i.e. the rule contains no free occurrences of term variables. The weakest-precondition transformer for such a rule is the composition of the weakest-precondition transformers of its components.

2.3 Strategies for control

2.3.1 The cut strategies—not, try and repeat

Three *Stratego* strategy constructors allow a compound strategy to succeed after an argument strategy has failed. The first such strategy requires failure of its argument; the other two always succeed if the argument strategy terminates. The corresponding logical rules are:

$$wp_{not(s)}(P) = P \wedge \overline{Dom(s)}$$

$$wp_{try(s)}(P) = wp_s(P) \vee (P \wedge \overline{Dom(s)})$$

$$\begin{aligned} wp_{repeat(s)}(P) &= \mu\xi.\ wp_s(\xi) \vee (P \wedge \overline{Dom(s)}) \\ &= \lim_{n\to\infty} \bigvee_{k=0}^{n} wp_{s^k}(P \wedge \overline{Dom(s)}) \end{aligned}$$

in which s^k denotes a k-fold repetition of the strategy s. To establish satisfiability of $wp_{repeat(s)}(P)$, one must demonstrate that there is a finite index k, for which $wp_{s^k}(P \wedge \overline{Dom(s)})$ is satisfied. It is necessary to show that the strategy terminates in order to establish that the *wp* formula is satisfiable.

2.4 CTL—a term logic

A logic capable of characterizing strategies must be able to express properties of the substructure of terms. For such a capability we turn to *Computational Tree Logic* (CTL)[3,6]. CTL is a modal logic conceived originally as a *branching-time* temporal logic. Nodes of a tree can be interpreted as the possible future states of a system as time progresses. The root of the tree represents the current state. Each path from the root represents a possible trajectory of the system being modeled.

However, characterizing possible future trajectories of a system is only one interpretation that can be made of a CTL formula. Its essential aspect is that it allows quantification of assertions independently along two dimensions of a tree—along a path, which may be finite or infinite, and across alternate paths, which are only finitely branching.

The along-paths, or depth quantifiers of CTL are **G**, read "globally", which quantifies (universally) over all subterms along a path that descends from the root of a tree, and **F**, read "eventually", which selects (existentially) a term somewhere along a path. Added to these is the specific along-path quantifier **X**, read "child", which selects the immediate subterm of the root along a given path.

The path, or breadth quantifier **A**, read "all paths", quantifies over all paths descending through a tree from its root, and **E**, read "some path", existentially selects a path from the root. Used together, the depth and breadth quantifiers allow one to express specific properties of a term and its subterms.

2.5 The modal mu-calculus

CTL modalities allow us to express logical formulas interpreted over terms, with separate quantifications over paths (breadth of a term) and levels (depth of a term). However, there are cases in which we should like to express depth quantification in a more detailed way. The μ-calculus [7] is a classical logic that provides least and greatest fixed-point binding operators (denoted by symbols μ and ν, respectively) well suited to expressing depth quantification. The *modal* μ-calculus is the basic μ-calculus enriched with the modal quantifiers $\mathbf{A}$ and $\mathbf{E}$ which express path quantification in a tree and the modal operator $\mathbf{X}$ to designate a property of immediate subterms.

Uses of the CTL depth quantifiers $\mathbf{G}$ and $\mathbf{F}$ can be expressed in terms of fixed-point expressions in the modal μ-calculus. For example, the CTL formula $\mathbf{AG}\,P$ (everywhere P) is logically equivalent to the formula $\mu\xi.\,P \wedge \mathbf{AX}\,\xi$ in the modal μ-calculus, and $\mathbf{EF}\,P$ (somewhere P) is equivalent to $\mu\xi.\,P \vee \mathbf{EX}\,\xi$. In the following sections, we shall use modal μ-calculus formulas to express weakest-preconditions of recursive strategies over terms.

2.6 Path quantification by term congruence

Path quantification can also be made more explicit by referring to paths directed through specific arguments of constructed terms. For example, a *Let* constructor (see Sec. 3) takes a triple of arguments, each of which is given a different interpretation in a language that embeds *Let* expressions. One might wish to quantify with respect to the last two arguments, omitting quantification over the first argument.

A mechanism that can express such selective quantification is *term congruence*, a device already employed to define strategies in the *Stratego* language. A term congruence lifts a term constructor to a constructor in another domain. For instance, the *Let* constructor has the signature

$$\mathit{Let} : \mathit{String} \ {*}\ \mathit{Expr} \ {*}\ \mathit{Expr} \to \mathit{Expr}$$

where *String* and *Expr* are sorts of type *Term*. When lifted to a domain of predicates, its signature becomes

$$\mathit{Let} : \mathit{Pred} \ {*}\ \mathit{Pred} \ {*}\ \mathit{Pred} \to \mathit{Pred}$$

Thus we can write $\mathit{Let}(P, P, P)$ to express the proposition $\mathit{Dom}(?\mathit{Let}(\_, \_, \_)) \wedge \mathbf{AX}\,P$. However, term congruence also permits one to express a property that is more specific with respect to paths, such as $\mathit{Let}(\mathit{True}, P, P)$, which asserts the predicate P over only the second and third subterms of a *Let* construction.

2.7 Weakest preconditions of non-local strategies

When a modal logic is interpreted over terms, a weakest precondition for success of a local strategy can be extended to characterize a global strategy whose effects occur throughout a term. For example, the strategy constructor *all*( ) applies a strategy s, given as its argument, to the children of a top-level term constructor and succeeds if and only if s succeeds at every one of the children. The weakest precondition for this strategy construction is expressed by

$$wp_{\mathit{all}(s)}(\mathbf{AX}\,P) = \mathbf{AX}\,(wp_s(P))$$

The weakest precondition for the strategy construction *some* to succeed on at least one term is

$$wp_{some(s)}(\mathbf{EX}\,P) = \mathbf{EX}\,(wp_s(P))$$

However, we often wish to make a stronger assertion about the result of applying a strategy constructed with *some*, one that accounts for its "greedy" nature. This is captured by the weakest precondition for a strategy *some*(s) to establish a condition P uniformly for all children of a node:

$$wp_{some(s)}(\mathbf{AX}\,P) = \mathbf{EX}\,(wp_s(P)) \;\wedge\; \mathbf{AX}\,(wp_s(P) \;\vee\; (P \;\wedge\; \overline{Dom(s)}))$$

A *bottom-up* strategy construction applies a strategy, s, to all subterms of a given term in bottom-up order. Thus an intermediate form might consist of a term, each of whose children had already been transformed by an application of *bottom-up*(s). A bottom-up strategy succeeds if and only if the argument strategy succeeds at every subterm. Its definition in *Stratego* is

```
bottom-up(s) = rec r(all(r); s)
```

Suppose the expected result of a bottom-up strategy is a term that satisfies a common property, P, throughout. A bottom-up strategy is characterized by a weakest-precondition defined as a least fixed-point:

$$wp_{bottom\text{-}up(s)}(\mathbf{AG}\,P) = \mu\xi.\,\mathbf{AX}\,\xi \;\wedge\; (\mathbf{AX}\,P \Rightarrow wp_s(P \;\wedge\; (\mathbf{AX}\,P)))$$

The implication expresses the condition that the common property P at every subterm is a sufficient precondition for the strategy s to succeed and establish the property P of the resulting term.

Analogously, a *top-down* strategy construction applies its argument to the subterms of a given term in top-down order. Its definition in *Stratego* is

```
top-down(s) = rec r(s; all(r))
```

Like a bottom-up strategy, a top-down strategy may also produce a result term characterized by a common property that holds throughout. The top-down strategy is characterized by:

$$wp_{top\text{-}down(s)}(\mathbf{AG}\,P) = \mu\xi.\,wp_s(\mathbf{AX}\,\xi \;\wedge\; (\mathbf{AX}\,P \Rightarrow P))$$

The strategy constructors *somebu* and *sometd* are similar, but only require the strategy application to children of a node to succeed on at least one, rather than all of the children. The *somebu*(s) and *sometd*(s) strategies succeed if there are one or more paths from the root of a term clear through to its fringe, along which the strategy s succeeds. Logical characterizations of these two strategies are:

$$wp_{somebu(s)}(\mathbf{EG}\,P) = \mu\xi.\,\mathbf{EX}\,\xi \;\wedge\; (\mathbf{EX}\,P \Rightarrow wp_s(P \;\wedge\; (\mathbf{EX}\,P)))$$

$$wp_{sometd(s)}(\mathbf{EG}\,P) = \mu\xi.\,wp_s(\mathbf{EX}\,\xi \;\wedge\; (\mathbf{EX}\,P \Rightarrow P))$$

To express that strategies *somebu*(s) or *sometd*(s) should produce a term with a property that holds everywhere, the weakest precondition must allow the possibility that the asserted property already holds in subterms on which the parameter strategy, s, does not succeed. For *sometd*, this is:

$$wp_{sometd(s)}(\mathbf{AG}\,P) = \mu\xi.\ wp_s(\mathbf{EX}\,\xi \wedge (\mathbf{EX}\,P \Rightarrow P) \wedge \mathbf{AX}\,(\xi \vee \mathbf{AG}\,P))$$

3 Example: Characterizing reduction strategies for lambda-calculus

As an example, let's consider reduction strategies for the lambda calculus with explicit substitution. An explicit substitution calculus affords more opportunities for control in reduction than does the calculus with implicit substitution. We begin with a signature for lambda terms, written in *Stratego*:

```
module lambda
signature
  sorts Expr
  constructors
    Var : String -> Expr
    Abs : String * Expr -> Expr
    App : Expr * Expr -> Expr
    Let : String * Expr * Expr -> Expr
```

Reduction rules of the calculus are given in the module, `lambda-rules`, printed below.

The rules `Alpha` and `Beta` are conversion/reduction rules of the lambda calculus. These rules suspend substitutions in the form of `Let` constructions. The *Stratego* library strategy `new` is a term-builder that upon each invocation, generates a new identifier not previously occurring in any term.

Rules `LetVar`, `LetApp` and `LetAbs` implement substitution of a given term for all free occurrences of a specified variable in a host term. These rules constitute a standard formulation of lambda-calculus with explicit substitution.

```
module lambda-rules
imports lambda lib
rules
  // lambda calculus rules
  Alpha : Abs(x,e) -> Abs(y,Let(x,Var(y),e))
                      where new => y

  Beta : App(Abs(x,m),n) -> Let(x,n,m)

  // Let distribution -- the substitution rules
  LetVar : Let(x,e,Var(x)) -> e
  LetVar : Let(x,e,Var(y)) -> Var(y)
           where <not(eq)> (x,y)

  LetApp : Let(x,e,App(m,n)) -> App(Let(x,e,m),Let(x,e,n))
  LetAbs : Let(x,e,Abs(x,m)) -> Abs(x,m)

  LetAbs : Let(x,e,Abs(y,m)) -> Abs(z,Let(x,e,n))
           where <Alpha> Abs(y,m) => Abs(z,n)
```

In the module `lambda-red`, we formulate three different strategies for reduction in the lambda calculus, using the set of rules given in `lambda-rules`. All use a common substitution strategy, `subst`, the first strategy declared in the module.

```
module lambda-red
imports lambda-rules
strategies
  // a strategy to eliminate Let constructions by forcing
  // substitution
  subst = rec r (Let(id,id,r)
                <+ sometd(LetVar + LetApp + LetAbs))

  // various strategies for reduction:
  left-outer = rec r (Beta + subst + App(r,id))

  all-outer  = rec r (App(id,r)
                     <+ Beta + Let(id,id,r)
                     <+ subst + App(r,id))

  reduce-all = rec r (App(id,r)  + Abs(id,r)
                     <+ Beta + Let(id,id,r)
                     <+ subst + App(r,id))
```

3.1 Properties of the substitution strategy

The strategy `subst` applies the `Let`-elimination rules top-down, pushing the `Let` construct deeper into terms until it can be eliminated by an instance of a `LetVar` or `LetAbs` rule. When the top-level expression is a `Let` construction on which none of the `Let`-elimination rules succeeds, the `subst` strategy applies itself recursively to the matrix of a `Let` term.

By recursively eliminating `Let`-terms nested within a `Let` construction, we avoid the need for an explicit `LetLet` rule to handle nested `Let` terms. The recursion is effective only in the matrix of a `Let` term. This strategy is consistent with outermost reduction of `Beta` redexes, but would not be consistent with innermost reduction. Thus the strategy is a bit subtle.

To give a weakest-precondition formula for the `subst` strategy we follow the outline of the *wp* formula for a top-down strategy. However, a general top-down strategy would apply path quantification over all paths, whereas in the `subst` strategy, the recursion is effective only over the particular paths specified by a term-congruence. To simplify the formulation, let's factor the substitution strategy so that the top-down application of `Let`-elimination rules becomes a strategy parameter.

```
let-elim = LetVar + LetApp + LetAbs

subst'(s) = rec r (Let(id,id,r) <+ s)

subst = subst'(sometd(let-elim))
```

The recursive strategy `subst'(s)` applies its parameter strategy, `s`, bottom-up in the matrix of possibly nested *Let*-terms. Thus `subst'(s)` can be effective on nested `Let` constructions.

$$Dom(subst'(s)) = \mu\xi.\, Let(True, True, \xi) \;\vee\; Dom(s)$$

In particular, when s is specialized to the strategy $sometd(let\text{-}elim)$, the domain can be seen to cover all possible forms of *Let* constructions:

$$\begin{aligned} Dom(subst'(sometd(let\text{-}elim))) = \mu\xi.\, & Let(True, True, \xi) \vee \\ & Let(True, True, Var(True)) \vee \\ & Let(True, True, App(True, True)) \vee \\ & Let(True, True, Abs(True, True)) \end{aligned}$$

Letting $NotLet =_{def} Var(True) \vee Abs(True, True) \vee App(True, True)$, we see that

$$\overline{Dom(subst'(sometd(let\text{-}elim)))} \equiv NotLet$$

Thus, the weakest-precondition under which an application of the strategy $subst'(sometd(let\text{-}elim))$ is assured to produce a let-free term can be shown to be

$$\begin{aligned} & wp_{subst'(sometd(let\text{-}elim))}(\mathbf{AG}\, NotLet) = \\ & \qquad \mu\xi.\, Let(True, True, \xi) \vee \\ & \qquad\quad (wp_{sometd(let\text{-}elim)}(\mathbf{AG}\, NotLet) \wedge \\ & \qquad\qquad \overline{Let(True, True, Dom(subst'(sometd(let\text{-}elim))))}) \end{aligned}$$

A more detailed characterization of $wp_{sometd(let\text{-}elim)}(\mathbf{AG}\, NotLet)$ has been omitted to save space.

3.2 Properties of the reduction strategies

Each of the three reduction strategies given in the module `lambda-red` has a different domain formula.

$$\begin{aligned} Dom(\mathit{left\text{-}outer}) = \mu\xi.\, & App(Abs(True, True), True) \vee \\ & Let(True, True, True) \vee \\ & App(\xi, True) \end{aligned}$$

$$\begin{aligned} Dom(\mathit{all\text{-}outer}) = \mu\xi.\, & App(True, \xi) \vee \\ & App(Abs(True, True), \neg\xi) \vee \\ & Let(True, True, True) \vee \\ & App(\xi, \neg\xi) \end{aligned}$$

$$\begin{aligned} Dom(\mathit{reduce\text{-}all}) = \mu\xi.\, & App(True, \xi) \ \vee\ Abs(True, \xi) \vee \\ & App(Abs(True, True), \neg\xi) \vee \\ & Let(True, True, True) \vee \\ & App(\xi, \neg\xi) \end{aligned}$$

The strategy *all-outer* reduces strictly more terms than does *left-outer* and *reduce-all* reduces more terms than does *all-outer*.

3.2.1 Normalization strategies

The module `normalize` contains normalization strategies that iterate the reduction strategies of module `lambda-red`. The *Stratego* library strategy `stdio` accepts input in textual format from the standard input file and delivers the output of its argument strategy to the standard output file.

```
module normalize
imports lambda-red io
strategies
  whnf = stdio(repeat(left-outer))

  hnf  = stdio(repeat(all-outer))

  normalize = stdio(repeat(reduce-all))
```

The normal forms achieved by each of the above strategies can be characterized in terms of modal μ-calculus formulas. The simplest to characterize is the `normalize` strategy. We first define a predicate to say what forms are β-redexes,

$$IsRedex = App(Abs(True, True), True)$$

Then a β-reduced normal form is one that contains no β-redex nor `Let` term,

$$\mathit{BetaNormal} = \mathbf{AG}\,(\neg \mathit{IsRedex} \wedge \mathit{NotLet})$$

Head-normal form is slightly more troublesome to define, as normalization is not required under an abstraction. To express this distinction, we need a two-place modality operator analogous to the "until" operator of linear temporal logic, which we here call **above**. Its definition as a modal μ-calculus formula is

$$P \ \mathbf{above}\ Q \equiv \mu\xi.\, Q \vee (P \wedge \mathbf{AX}\,\xi)$$

Then head-normal form is

$$\mathit{HeadNormal} =$$

$$(\neg \mathit{IsRedex} \wedge \mathit{NotLet})\ \mathbf{above}\ (\mathit{Abs}(\mathit{True}, \mathit{True}) \wedge \mathbf{AG}\,\mathit{NotLet})$$

Finally, to express weak head-normal form, which is the expected form of terms normalized by the `whnf` strategy, we resort to term-congruence operators to specify selective path quantification;

$$\mathit{WeakHead} =$$

$$(\mu\xi.\,(\mathit{Var}(\mathit{True}) \vee \mathit{App}(\xi, \mathit{True})))\ \mathbf{above}\ (\mathit{Abs}(\mathit{True}, \mathit{True}) \wedge \mathbf{AG}\,\mathit{NotLet})$$

Notice that in the latter formula, the embedded fixed-point formula describes only the condition in the prefix of the **above** operator, it does not encompass the entire geography of a term that contains embedded abstractions.

4 Other transformation systems—related work

The antecedent of all strategy languages is the prototypical ML language designed by Robin Milner to support proof construction in LCF [10]. In the last decade, new languages have evolved, reflecting lessons learned from logic programming on the one hand, and on the other, from understanding and implementing efficient term rewriting as a computational paradigm. Maude [8] implements a rewriting logic based upon a theory of term equality relative to an environment. Maude does not cater explicitly for programming strategies but supports strategy programming via reflection in the language [9].

ASF+SDF [11] is a general-purpose language to support term manipulation and has been used for the construction of parsers and pretty-printers as well as transformations. It does not support strategy-controlled rewriting directly, but a notion of traversal strategies can be implemented in this programming environment.

Strategies to control rewriting were introduced in ELAN [16], a comprehensive TRS with support for commutative and associative-commutative rewriting. ELAN employs reflection in the language [2], allowing strategies themselves to be expressed in terms of rewrite rules, although several specific strategy constructions have been built-in.

In ELAN, the idea of programming strategies for term rewriting was made an explicit goal [1]. ELAN has experimented with three primitives for controlling choice among possible alternate strategies: left-biased choice **case**, nondeterministic committed choice (called **dc**, for "don't care") and nondeterministic choice by consequence (called **dk**, for "don't know") which requires either backtracking or an equivalent implementation mechanism. In designing *Stratego*, choice-by-consequence has been rejected as computationally expensive and rarely needed in practice. Instead, the *Stratego* programmer is expected to anticipate the consequences of alternatives and specify an ordering of choices when the domains of alternative strategies may overlap.

ELAN is formally defined by a denotational semantics [1] which provides a reference model for implementation. In principle, the semantics also furnishes a basis for reasoning about transformation strategies. However, reasoning directly in terms of a semantics model can be tedious, as it is encumbered by details of the model.

5 Conclusions

The contribution of this paper lies in showing that two computational logics, each developed for a somewhat different purpose, can be used in combination to yield a programming logic for term transformation strategies, a domain for which no completely satisfactory logical characterization had previously been developed.

Weakest-precondition logic was originally proposed by Edsger Dijkstra [4] to cope with problems arising from nondeterministic choice, concurrency and potential nontermination of programs. Analogous problems arise when attempting to characterize properties of transformation strategies.

Strategies incorporate control to program traversals over complex terms in a variety of ways. CTL and the modal μ-calculus were originally developed for temporal applications in which paths in terms are thought of as evolving through time. However, these formalisms are equally applicable to terms whose paths are spatial. These notations provide the ability to quantify separately over paths (the breadth of a term), or over depth in a term. We have added to the generic quantification, operations that specify path quantification by lifting the constructors of terms to the status of logical quantifiers, an implicit term congruence.

The vehicle for this investigation into transformation strategies is *Stratego*, a domain-specific language that inherits from both logic and functional programming traditions. *Stratego* provides a compositional approach to programming strategies for term transformation that was lacking in earlier systems.

Acknowledgments: The author is grateful to Eelco Visser for conversations on transformation strategies and their formulation in *Stratego*, and to

Tony Hoare, for a patient explanation of the subtleties of weakest precondition logics.

References

[1] Peter Borovansky. *Le controle de la reecriture: etude et implantation d'un formalisme de strategies.* PhD thesis, Universite Henri Poincare–Nancy I, October 1998.

[2] Peter Borovansky, Claude Kirchner, and Helene Kirchner. Controlling rewriting by rewriting. *Electronic Notes in Theoretical Computer Science*, 5, 1997.

[3] E. M. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.

[4] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[5] X. Du, K McDonnell, E. Nanos, Y. S. Ramakrishan, and Scott A. Smolka. Software design, specification and verification: Lessons learned from the Rether case study. In *Proc. of Sixth International Conference on Algebraic Methods and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 185–198. Springer Verlag, 1997.

[6] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronizations skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[7] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[8] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 5. Elsevier, September 1996.

[9] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 5. Elsevier, September 1996.

[10] Robin Milner, Mike Gordon, and Christopher Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science.* Springer Verlag, 1979.

[11] M. G. J. van den Brand, S. M. Eijkelkamp, D. K. A. Geluk, H. Meijer, M. J. F. Polling, and H. R. Osburne. Program transformation using ASF+SDF. Technical Report P9504, Programming Research Group, University of Amsterdam, 1995.

[12] Eelco Visser. A bootstrapped compiler for strategies. In B. Gramlich, H. Kirchner, and F. Pfenning, editors, *Strategies in Automated Deduction (STRATEGIES'99)*, pages 78–83, July 1999.

[13] 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. Springer Verlag, July 1999.

[14] Eelco Visser and Zine el Abidine Benaissa. A core language for rewriting. In Claude Kirchner and Helene Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, Electronic Notes in Theoretical Computer Science. Elsevier, September 1998.

[15] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proc. of 1998 ACM/SIGPLAN International Conference on Functional Programming*, pages 13–26. ACM Press, September 1998.

[16] M. Vittek. A compiler for nondeterministic rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*. Springer Verlag, July 1996.