*Electronic Notes in Theoretical Computer Science 58 No. 2 (2001)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume58.html*](http://www.elsevier.nl/locate/entcs/volume58.html) *17 pages*

*A Logic for Rewriting Strategies Richard B. Kieburtz* [*1*](#_bookmark0) *;*[*2*](#_bookmark0)

*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 stra- tegy language Stratego. This logic embeds the modal mu-calculus, allowing it to express properties of terms of arbitrary depth. Its use is illustrated by characteri- zing 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 bin- dings 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 ea- sier with appropriate programming language support and better understood through logical characterization.

*This paper is a rst step towards de ning a programming logic for strate-* gies. 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 -calculus [[7],](#_bookmark8) enriched with modalities that express path quanti cation 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.*

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license*](http://creativecommons.org/licenses/by-nc-nd/3.0/)*.*

*Rules have been developed in this logic for the constructions of Stratego* [[12](#_bookmark13)[,13,15,14](#_bookmark15)], a domain-speci c language designed speci cally for program- ming strategies. Stratego provides a compositional semantics with explicit recursion, allowing strategies to be applied at sites deep within terms. Stra- tegies, and therefore patterns, are rst-class constructs of Stratego. As in a logic programming language, conditional control in Stratego is based upon the success or nite failure of strategies, rather than if-then-else expressions that test conditions coded as boolean values. Although essentially a rst-order language, Stratego also supports a particular form of higher-order strategies, namely term congruences, which lift term constructors into strategy construc- tors.

*Section* [*2*](#_bookmark1) *introduces a weakest-precondition (wp) logic for strategies. In* Section [3](#_bookmark2), 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 re- writing, 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 speci ed form. A predicate characterizing the largest set of such input terms is a weakest precondition for the strategy to produce a speci ed form of output.

*In a weakest-precondition logic, each rewrite rule or strategy is characteri-* zed 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 nite signature, . Call this universe T ( ). Predicates are interpreted as subsets of this universe, by an interpretation function, I : Pred ! T ( ). The distinguished predicates True and False have the interpretations I(True) = T ( ) and I(False) = ;. Term variables in the logic range over the universe.

*Connectives (:), (\_), (^) and ()) are used to form compound predicate* formulas. They have the interpretations

*I(:P )= ft j t 62 I(P )g I(P \_ Q)= I(P ) [ I(Q)*

*I(P ^ Q)= I(P ) \ I(Q)*

*I(P ) Q)= I(P ) I(Q)*

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

*If s is a strategy for term rewriting, then by wps : Pred ! Pred we denote* the predicate transformer associated to s. The expression hsi t denotes the

*application of strategy s to term t. We denote by Dom(s) the set ft j hsi t 2*

*I(True)g. 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 nite failure of a strategy is used for control, whereas failure by nontermination obviously cannot be. Denote by Dom(s) the set on which strategy s fails nitely. In case strategy s terminates uniformly, Dom(s) \_ Dom(s) = True.

*2.1 Rules of wp logic for Stratego*

*The weakest-precondition logic satis es several rules, including*

*P ) Q*

*wps(P ) ) wps(Q)*

*wps(True) = Dom(s) wps(False) = False wps;t(P ) = wps(wpt(P ))*

*wps<+t(P ) = wps(P ) \_ (wpt(P ) ^ Dom(s))*

*wps+t(P ) = (wps(P ) ^ Dom(t)) \_ (wpt(P ) ^ Dom(s)) wps(:P ) ^ wps(P ) = False*

*The rules for alternatives combinators (+) and (<+) are those of committed*

*choice. With committed choice, a potential choice strategy will not be e ective* 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 speci ed postcondition. However, this is a constructive interpretation. If t 2 wps(P ), then the strategy application hsit is accepted as constructive evidence of a term that satis es P . Thus it should come as no surprise that the characterization

*it gives for nondeterministic choice (+) is rather weak. For example,*

*wps+s(P )= (wps(P ) ^ Dom(s)) \_ (wps(P ) ^ Dom(s))*

*= wps(P ) ^ Dom(s)*

*= False*

*Since the strategy s + s speci es a nondeterministic choice between two stra-* tegies 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 pre-* condition, because it speci es a domain on which the component strategies of a choice can be discriminated. The weakest discriminating precondition wpr+s(P ) can supply de nite information only over a domain on which stra-

*tegies r and s both terminate but cannot both succeed.*

*Although it is tempting to give a more optimistic interpretation of non-* deterministic choice, such an interpretation could give rise to diÆculties. In

*a so-called angelic interpretation of nondeterminism, wps+s(P ) = wps(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 transfor-* med. 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 provi- ding an environment in which its variables are bound. A strategy can have the e ect 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, [(x1; t1); (x2; t2);::: ;* (xn; tn)], in which x1; x2;::: ; xn designate variables and t1; t2;::: ; tn are ground terms. We shall write [(x; t) j 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 environ-* ment. Axioms of the judgment form include the usual axioms of equality, plus

*[(x; t) j e] j= x = t : True*

*e j= x = t : True [(y; t0) j e] j= x = t : True (y 6= x)*

*We say that an environment e1 re nes an environment e2, writing e1 e2,* if e1 is compatible with e2 on all bindings visible in e2, but it may also contain additional bindings.

*e1 e2 =def 8x; t: e2 j= x = t : True ) e1 j= x = t : 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, t;e denotes the characteristic pre- dicate of a set comprised of a single pair, f(t; e)g. The weakest precondition for a predicate P to hold of the result of a strategy, s is de ned in terms of a predicate transformer, wp0 , restricted to the transformation of singleton pre- dicates. This elementary predicate transformer can be de ned directly with term equality judgments for each primitive strategy.

*s*

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

*Admissible(P ) () wp (P ) = f(t; e) j 9t1: (wp0 ( t ;;) ) t;e) ^ ; j= t1 : P g*

*s s 1*

*The above de nition allows us to calculate the weakest precondition of a* composition of two strategies:

*wpr;s(P )= wpr(wps(P ))*

*= wpr( : f(t; e) j 9t1:(wp0 ( t ;;) ) t;e) ^*

*s 1*

*; j= t1 : P ^ e j= t : g)*

*= : f(t; e) j 9t1; t2; e2: (wp0 ( t ;e ) ) t;e) ^*

1. *2 2*

*(wp0 ( t ;;) ) t ;e ) ^*

1. *1 2 2*

*; j= t1 : P ^ e j= t : g*

*in which designates the least xed-point binding operator of the -calculus* [[7](#_bookmark8)].

*This wp composition rule shows how a composition of strategies propagates* re nements 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 bin- dings found in the current environment. Predicate transformations for the elementary strategies of pattern-matching and term building are:

*wp?t(P ) = f(t0;e j 9e0: e0 e ^ e0 j= t0 = t : P g*

*wp!t(P ) = f(t0; e) j e j= t : P g*

*A pattern strategy succeeds if an initial term and the given pattern can be* matched by instantiating variables. When it succeeds, it produces a re nement 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 satis es 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*

*wptest(s)(P ) = f(t; e) j (t; e) 2 wps(True) ^ e j= t : P g*

*2.2.3 Restricting the scope of variables*

*New variables may be introduced in the scope of a particular strategy. The* notation [(x; ?) j 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

*wpfx1 ;:::;xm : sg(P ) = f(t; e) j (t; [(x1; ?);::: ; (xm; ?) j e]) 2 wps(P )g*

*2.2.4 Derived strategies*

*Many complex strategies can be de ned in terms of these basic binding and* building strategies. For instance, a strategy that applies a given strategy, s to a speci c term is !t; s. This strategy can be written in a function-application style with the syntax hsi t.

*Another example is a rule in Stratego, which has the (sugared) syntax* n p ! t where r n . A rule is de ned in terms of the compound strategy fx1;::: ; xng : ?p; r; !t, with the side condition that FV (p) [ FV (r) [ FV (t) fx1;::: ; xng, 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 rst such strategy requires fai- lure of its argument; the other two always succeed if the argument strategy

*terminates. The corresponding logical rules are:*

*wpnot(s)(P ) = P ^ Dom(s)*

*wptry(s)(P ) = wps(P ) \_ (P ^ Dom(s))*

*wprepeat(s)(P )= : wps( ) \_ (P ^ Dom(s))*

*= lim !1 Wn wp (P ^ Dom(s))*

*n k=0 sk*

*in which sk denotes a k-fold repetition of the strategy s. To establish satis-*

*ability of wprepeat(s)(P ), one must demonstrate that there is a nite index* k, for which wpsk (P ^ Dom(s)) is satis ed. It is necessary to show that the strategy terminates in order to establish that the wp formula is satis able.

*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 Computa- tional Tree Logic [(CTL)[3](#_bookmark5)[,6].](#_bookmark7) 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 quanti cation of assertions independently along two dimensions of a tree|along a path, which may be nite or in nite, and across alternate paths, which are only nitely branching.

*The along-paths, or depth quanti ers of CTL are G, read \globally", which* quanti es (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 speci c along-path quanti er X, read \child", which selects the immediate subterm of the root along a given path.

*The path, or breadth quanti er A, read \all paths", quanti es over all* paths descending through a tree from its root, and E, read \some path", exi- stentially selects a path from the root. Used together, the depth and breadth quanti ers allow one to express speci c 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 quanti cations over paths (breadth of a term) and levels (depth of a term). However, there are cases in which we should like to express depth quanti cation in a more detailed way. The -calculus [[7](#_bookmark8)] is a classical logic that

*provides least and greatest xed-point binding operators (denoted by symbols*

*and , respectively) well suited to expressing depth quanti cation. The* modal -calculus is the basic -calculus enriched with the modal quanti ers A and E which express path quanti cation in a tree and the modal operator

*X to designate a property of immediate subterms.*

*Uses of the CTL depth quanti ers G and F can be expressed in terms of*

*xed-point expressions in the modal -calculus. For example, the CTL formula* AG P (everywhere P ) is logically equivalent to the formula : P ^ AX in the modal -calculus, and EF P (somewhere P ) is equivalent to : P \_ EX . In the following sections, we shall use modal -calculus formulas to express weakest-preconditions of recursive strategies over terms.

*2.6 Path quanti cation by term congruence*

*Path quanti cation can also be made more explicit by referring to paths di-* rected through speci c arguments of constructed terms. For example, a Let constructor (see Sec. [3)](#_bookmark2) takes a triple of arguments, each of which is given a di erent interpretation in a language that embeds Let expressions. One might wish to quantify with respect to the last two arguments, omitting quanti ca- tion over the rst argument.

*A mechanism that can express such selective quanti cation is term congru-* ence, a device already employed to de ne 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

*Let : String \* Expr \* Expr ! Expr*

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

*Let : Pred \* Pred \* Pred ! Pred*

*Thus we can write Let(P; P; P ) to express the proposition Dom(?Let( ; ; )) ^* AX P . However, term congruence also permits one to express a property that is more speci c with respect to paths, such as Let(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 e ects 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

*wpall(s)(AX P ) = AX (wps(P ))*

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

*wpsome(s)(EX P ) = EX (wps(P ))*

*However, we often wish to make a stronger assertion about the result of ap-* plying 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:

*wpsome(s)(AX P ) = EX (wps(P )) ^ AX (wps(P ) \_ (P ^ 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 de nition in Stratego is

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

*Suppose the expected result of a bottom-up strategy is a term that satis es* a common property, P , throughout. A bottom-up strategy is characterized by a weakest-precondition de ned as a least xed-point:

*wpbottom-up(s)(AG P ) = : AX ^ (AX P ) wps(P ^ (AX P )))*

*The implication expresses the condition that the common property P at every* subterm is a suÆcient 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 de nition 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:

*wptop-down(s)(AG P ) = : wps(AX ^ (AX P ) 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:

*wpsomebu(s)(EG P ) = : EX ^ (EX P ) wps(P ^ (EX P )))*

*wpsometd(s)(EG P ) = : wps(EX ^ (EX P ) 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:

*wpsometd(s)(AG P ) = : wps(EX ^ (EX P ) P ) ^ AX ( \_ 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 a ords more opportu- nities 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 identi er not previously occurring in any term.

*Rules LetVar, LetApp and LetAbs implement substitution of a given term* for all free occurrences of a speci ed variable in a host term. These rules con- stitute 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 di erent strategies for re-* duction in the lambda calculus, using the set of rules given in lambda-rules. All use a common substitution strategy, subst, the rst 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 e ective 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 quanti cation over all paths, whereas in the subst strategy, the recursion is e ective only over the particular paths speci ed 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 e ective on nested Let constructions.

*Dom(subst0(s)) = : Let(True; True; ) \_ Dom(s)*

*In particular, when s is specialized to the strategy sometd(let-elim), the do-* main can be seen to cover all possible forms of Let constructions:

*Dom(subst0(sometd(let-elim))) = : Let(True; True; ) \_*

*Let(True; True; Var(True)) \_*

*Let(True; True; App(True; True)) \_* Let(True; True; Abs(True; True))

*Letting NotLet =def Var(True) \_ Abs(True; True) \_ App(True; True), we see* that

*Dom(subst0(sometd(let-elim))) NotLet*

*Thus, the weakest-precondition under which an application of the strategy* subst0(sometd(let-elim)) is assured to produce a let-free term can be shown to be

*wpsubst0 (sometd(let-elim))(AG NotLet) =*

*: Let(True; True; ) \_*

*(wpsometd(let-elim)(AG NotLet) ^*

*Let(True; True; Dom(subst0(sometd(let-elim)))))*

*A more detailed characterization of wpsometd(let-elim)(AG NotLet) has been omit- ted to save space.*

*3.2 Properties of the reduction strategies*

*Each of the three reduction strategies given in the module lambda-red has a* di erent domain formula.

*Dom(left-outer) = : App(Abs(True; True); True) \_*

*Let(True; True; True) \_* App( ; True)

*Dom(all-outer) = : App(True; ) \_*

*App(Abs(True; True); : ) \_ Let(True; True; True) \_ App( ; : )*

*Dom(reduce-all) = : App(True; ) \_ Abs(True; ) \_*

*App(Abs(True; True); : ) \_ Let(True; True; True) \_ App( ; : )*

*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 re-* duction strategies of module lambda-red. The Stratego library strategy stdio accepts input in textual format from the standard input le and delivers the output of its argument strategy to the standard output le.

*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 charac-* terized in terms of modal -calculus formulas. The simplest to characterize is the normalize strategy. We rst de ne a predicate to say what forms are

*-redexes,*

*IsRedex = App(Abs(True; True); True)*

*Then a -reduced normal form is one that contains no -redex nor Let term,* BetaNormal = AG (:IsRedex ^ NotLet)

*Head-normal form is slightly more troublesome to de ne, 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 de nition as a modal -calculus formula is

*P above Q : Q \_ (P ^ AX ) Then head-normal form is*

*HeadNormal =*

*(:IsRedex ^ NotLet) above (Abs(True; True) ^ AG 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 quanti cation;

*WeakHead =*

*( : (Var(True) \_ App( ; True))) above (Abs(True; True) ^ AG NotLet)*

*Notice that in the latter formula, the embedded xed-point formula describes* only the condition in the pre x 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].](#_bookmark11) In the last decade, new languages have evolved, re ecting lessons learned from logic programming on the one hand, and on the other, from understanding and implementing eÆcient term rewriting as a computational paradigm. Maude

[*[8*](#_bookmark9)*] 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 re ection in the language [[9].](#_bookmark10)

*ASF+SDF* [*[11]*](#_bookmark12) *is a general-purpose language to support term manipu-* lation 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 pro- gramming environment.

*Strategies to control rewriting were introduced in ELAN* [*[16],*](#_bookmark17) *a compre-* hensive TRS with support for commutative and associative-commutative re- writing. ELAN employs re ection in the language [[2],](#_bookmark4) allowing strategies themselves to be expressed in terms of rewrite rules, although several speci c strategy constructions have been built-in.

*In ELAN, the idea of programming strategies for term rewriting was made* an explicit goal [[1].](#_bookmark3) ELAN has experimented with three primitives for con- trolling choice among possible alternate strategies: left-biased choice case, nondeterministic committed choice (called dc, for \don't care") and nonde- terministic 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 expen- sive and rarely needed in practice. Instead, the Stratego programmer is ex- pected to anticipate the consequences of alternatives and specify an ordering of choices when the domains of alternative strategies may overlap.

*ELAN is formally de ned by a denotational semantics* [*[1]*](#_bookmark3) *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 di erent 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]*](#_bookmark6) to cope with problems arising from nondeterministic choice, concurrency and potential nontermination of programs. Analogous problems arise when att- empting to characterize properties of transformation strategies.

*Strategies incorporate control to program traversals over complex terms* in a variety of ways. CTL and the modal -calculus were originally deve- loped 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 quanti cation, operations that specify path quanti-

*cation by lifting the constructors of terms to the status of logical quanti ers,*

*an implicit term congruence.*

*The vehicle for this investigation into transformation strategies is Stra-* tego, a domain-speci c language that inherits from both logic and functio- nal 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 conver-* sations 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, speci cation and veri cation: 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 -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. Re ection 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.*