More Precise Typing of Rewrite Strategies

Azamat Mametjanov
University of Nebraska at Omaha
USA
amametjanov@unomaha.edu

Victor Winter
University of Nebraska at Omaha
USA
vwinter@unomaha.edu

Ralf Lämmel
University of Koblenz-Landau
Germany
rlaemmel@acm.org

ABSTRACT

The programming concept of rewrite strategies supports versatile composition of rewrite rules and control of their application. Such programmability of rewrites can possibly lead to incorrect compositions of rewrites or incorrect applications of rewrites to terms within a strategic rewriting program. In this paper, we explore the analysis of strategic rewriting programs to detect certain programming errors statically. In particular, we introduce fine-grain types to closely approximate the dynamic behavior of rewriting. We develop an expressive type system for a core rewriting language. The resulting system detects programming errors of universally unreachable and failing rewrites. Static detection of such errors can substantially reduce testing and debugging efforts and lead to a more effective use of strategic rewriting in large and complex rewriting problems.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1.1 [**Programming Techniques**]: Functional Programming; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design, Languages, Theory

Keywords

Types, type-checking, transformation, term rewriting, strategies

1. INTRODUCTION

Term rewriting [1] is a computational model, similar to functional programming, that has proven itself useful in a variety of applications including equational reasoning, symbolic computation, language semantics and program transformation.

The classical (term) rewriting system consists of a set of rewrite rules that is *convergent*, that is, both *confluent* and *terminating*. However, when using rewriting in diverse programming contexts related to software analysis and transformation, rule sets arise that are neither confluent nor terminating. Under these circumstances, in order to make effective use of rewriting, explicit mechanisms must be introduced capable of exercising control over reduction sequences giving rise to rewrite strategies. This idea has been exercised in various strategic programming frameworks with different kinds of composition operators; see, e.g., [15, 2, 14, 17].

The explicit control in such framework can and should be subjected to analysis that determines whether a strategic program is semantically well-formed. For example, a sub-strategy embedded in a larger strategy in such a manner that it can never be applied (i.e., it is dead code) is ill-formed. Similarly, a sub-strategy that can be reached but cannot be successfully applied to any term is also ill-formed. In practice, ill-formed strategies are the Achilles heel of strategic programming and present a major obstacle to the effective application of strategic programming to large complex problems.

Contributions of the paper

The overall contribution of this research is the formulation of a decidable type system which takes into account the implications of (pattern-) matching so that more precise strategy types are used. To this end, the type system features the following ingredients:

- The constructors of user-defined datatypes are incorporated into the type system itself. As a result, ground terms and patterns (i.e., terms containing matchable variables) become types in their own right. Hence, types of strategies (e.g., rewrite rules) become more precise.

- The conditional and sequential composition of user-defined strategies is incorporated into the type system itself. As a result, the different possible argument and result patterns of a composed strategy are modelled precisely. Hence, applications of strategies to terms may be revealed as admitting no outcome at all.

In this paper, we limit our discussion to a small subset of strategic programming constructs. In particular, we do not include recursion, traversals and rule conditions. However, we provide some insights at the end of the paper in regards to the extension of the presented analysis onto these features.

Road-map of the paper

Our presentation is organized as follows. In Section 2 we briefly review the core abstractions in the programming of rewrite strategies. Section 3 discusses the common sources of errors that occur in strategic rewriting. Sections 4 and 5 present the essential aspects of type analysis and its properties. Section 6 relates this work to previous research. Finally, Section 7 concludes the paper.[1]

2. BACKGROUND

We begin with a brief overview of strategic rewriting: in particular rewrite rules, notions of success and failure of rewrites and strategic compositions of rewrite rules. We choose Standard ML (SML) [13] as the embedding language of rewrite strategies, however our discussion is sufficiently broad to include implementations of rewrite strategies in other languages such as Haskell [11], Stratego [15] and TL [17].

```
datatype factor  = Var of string          | Num of int;
datatype term    = Mult of term * factor  | Term of factor;
datatype expr    = Plus of expr * term    | Expr of term;

datatype ('i, 'o) options  = Some of 'o | None of 'i ; (* not NONE *)
type     ('i, 'o) strategy = 'i -> ('i, 'o) options;
```

Figure 1: Arithmetic expressions and strategy types

Consider the language of arithmetic expressions defined by the grammar in Fig. 1. Conceptually, terms of this language are additions and multiplications of variables and integers. To define rewrite rules over terms of this language, we can use SML functions. For instance, the following functions encode two rewrite rules $Expr[\![“a”]\!] \rightarrow Expr[\![“b”]\!]$ and $Expr[\![“b”]\!] \rightarrow Expr[\![“c”]\!]$:

```
fun r1 (Expr(Term(Var “a”))) = Some (Expr(Term(Var “b”)))
  | r1 t                     = None t;
val r1 : expr → (expr,expr) options
```
[2]

```
fun r2 (Expr(Term(Var “b”))) = Some (Expr(Term(Var “c”)))
  | r2 t                     = None t;
val r2 : expr → (expr,expr) options
```

The type of these functions adheres to the typing scheme of an *('i,'o) strategy* in Fig. 1, where the type variables range over input and output terms manipulated by the rewriting: here *(expr, expr) strategy*. The return type of the functions makes explicit not only the success or failure of an application but also the term that caused a failure, which is used by some term traversals. We use this design to unify both kinds of strategic rewriting semantics: (a) *Stratego-style* [15], where application failure produces the meta-term $\uparrow$ (or *Nothing* in Haskell) and (b) *TL-style* [17], where application failure produces a tuple $\langle t, false\rangle$.

Functions below encode the standard primitive rewrite abstractions *id*, which always succeeds on any term and *fail*, which always fails on all terms:

```
fun id t = (Some t): ('a,'a) options
val id = fn : 'a → ('a,'a) options          (* succeed on any term *)

fun fail t = None t
val fail = fn : 'a → ('a,'b) options          (* fail on any term *)
```

[1]The paper's web-page provides access to all of the source code used in the paper:
http://myweb.unomaha.edu/~amametjanov/stratcore

[2]We use this font to denote SML compiler- and interpreter-generated output.

Observability of application failure is the hallmark of strategic rewriting. It enables explicit handling of rewrite failures. In other words, a programmer can strategically manage the rewriting by attempting another rule(s) on the term that caused the failure. This algorithmic style of rewriting is provided by the conditional and sequential composition combinators:

```
fun choice s1 s2 t =              fun sequ s1 s2 t =
  case (s1 t) of                    case (s1 t) of
    Some t1 ⇒ Some t1                 Some t1 ⇒ s2 t1
  | None t1 ⇒ s2 t1;                | None t1 ⇒ None t1;
```

The operational behavior of the two combinators can be observed on the following application tests summarized as a semi-exhaustive truth table:

Strategy	Term	Application Result
(choice r1 r2)	(Expr(Term(Var “a”)))	Some (Expr(Term(Var “b”)))
(choice r1 r2)	(Expr(Term(Var “b”)))	Some (Expr(Term(Var “c”)))
(choice r1 r2)	(Expr(Term(Var “c”)))	None (Expr(Term(Var “c”)))
(sequ r1 r2)	(Expr(Term(Var “a”)))	Some (Expr(Term(Var “c”)))
(sequ r1 r2)	(Expr(Term(Var “b”)))	None (Expr(Term(Var “b”)))

3. CATEGORIES OF ERRORS

Having defined the primary abstractions of strategic rewriting, let us now consider some of the errors that often arise in the development of program transformations. While one kind of errors in Section 3.3 has been previously identified in [10], our approach in Section 4 provides a more precise solution. A unifying theme among all of these errors is the insufficient precision of the static type analysis provided by the type system – in our case SML's type system. For rewrite strategy implementations that do not use the strong typing of functional programming, the hazards of making errors are even greater.

3.1 Application Errors

Consider a simplified task of removing additions with zero. The first attempt toward this end could be the following strategy:

```
fun addZero (Plus(Expr(Term(Num 0)), pat)) = Some (Expr pat)
  | addZero t                              = None t;
val addZero = fn : expr → (expr, expr) options
```

Suppose that this strategy is applied to some terms within a program. This strategy is quantified over terms of type *expr* and therefore any application to a non-*expr* term is statically flagged as a type error. However, the strategy is well-typed for *any expr* term including terms that do not match the first input pattern. For example, a programmer might apply the strategy to a similar (abstract) term $x + 0$ expecting to obtain *Some* x. However, run-time tests would reveal that such application actually produces $None\ (x+0)$: e.g.

```
addZero (Plus(Expr(Term(Var “x”)), Term(Num 0)));
val it = None (Plus (Expr (Term (Var "x")),Term (Num 0)))
       : (expr, expr) options
```

What we would like to have happened is for the type system to statically reject such application because it cannot succeed and always produces a strategic option *None t*. This brings us to the definition of a strategic type error, which extends the standard notion of a type error:

Strategic type error: Application of a strategy to a term that *always* fails.

While it is clearly helpful to have a type system that rejects ill-typed applications, it would be even more beneficial if the type system could reject the strategic type errors as well. Such system would reduce testing and debugging by statically detecting applications that always fail and help a programmer in developing rewrites that can actually succeed.

Having identified another case of addition with a zero, where the zero appears on the right, let us now handle this case. In the spirit of strategic handling of failures, we will conditionally compose the rewrite *addZero* with a new rewrite that acts on the missing case. In the functional incarnation of strategic rewriting, this could be accomplished by extending the function *addZero* with a new input pattern. However, we choose the conditional combinator *choice* to remain in the overall programming style of one function per rewrite rule.

```
fun addZero' (Plus(Expr x, Term(Num 0))) = Some (Expr x)
  | addZero' t                           = None t;
val addZero' = fn : expr → (expr, expr) options

val addZeroes = choice addZero addZero'
val addZeroes = fn : expr → (expr, expr) options
```

3.2 Sequential Composition Errors

In a sequential composition, two strategies are applied on an input term in sequence such that the second strategy is applied on the output term of the first strategy. Such composition fails if one or both of the strategies fail. If the second strategy cannot succeed on the output of the first strategy, then a strategic type error occurs because the composition will always fail at run-time.

For example, consider the simplification of multiplications. First, we simplify multiplications with the absorbing element 0:

```
fun timesZero (Mult(Term(Num 0), _ )) = Some (Term (Num 0))
  | timesZero t                       = None t;
val timesZero = fn : term → (term, term) options

fun timesZero' (Mult(_ , Num 0))      = Some (Term (Num 0))
  | timesZero' t                      = None t;
val timesZero' = fn : term → (term, term) options

val timesZeroes = choice timesZero timesZero'
val timesZeroes = fn : term → (term, term) options
```

Next, we simplify multiplications with the identity element 1:

```
fun timesOne (Mult(Term(Num 1), x)) = Some (Term x)
  | timesOne t                      = None t;
val timesOne = fn : term → (term, term) options

fun timesOne' (Mult(Term x, Num 1)) = Some (Term x)
  | timesOne' t                     = None t;
val timesOne' = fn : term → (term, term) options

val timesOnes = choice timesOne timesOne'
val timesOnes = fn : term → (term, term) options
```

And now, to combine the two simplifications together we could try to sequentially compose the two conditional compositions:

```
val simpTimes = sequ timesZeroes timesOnes;
val simpTimes = fn : term → (term, term) options
```

Note that as far as SML's type system is concerned, the composition is well-typed as indicated by the system-generated type. However, the composition will always fail at run-time for all possible input terms. This is due to the fact that the successful outputs of the strategy *timesZeroes*—*Some (Term (Num 0))*—do not contain multiplications within them and therefore both *timesOne* and *timesOne'* will fail leading to the overall simplification failure.

A sequencing of strategies that does not have a strategic type error is the reverse of the first attempt:

```
val simpTimes' = sequ timesOnes timesZeroes;
val simpTimes' = fn : term → (term, term) options
```

A closer (and manual) inspection and/or testing of this composition would reveal that it can only succeed on terms like *x * 1 * 0* that have both kinds of simplification patterns, but not on terms like *x * 1*. One of the indirect benefits of a type system is the feedback to a programmer in the form of the computed type, which could be compared to an expected type for consistency. In this respect, the type *(term, term) options* is too coarse to reveal the over-constrained specificity of expected inputs to this strategy. A type system that can detect strategic type errors could automatically compute fine-grain types that would indicate that expected input terms were of the form *x * 1 * 0*, which would give the programmer the opportunity of static detection of the mismatch between what the programmer wanted to do and what actually was programmed.

Thus, a less constrained and more appropriate combination is the conditional composition:

```
val simpTimes" = choice timesOnes timesZeroes;
val simpTimes" = fn : term → (term, term) options
```

3.3 Conditional Composition Errors

In a conditional composition, the second strategy is attempted on the input term if the first strategy fails on the term. A strategic error occurs when the second strategy cannot succeed because it is subsumed by the first strategy. In other words, the second strategy becomes unreachable (or dead) code.

For example, consider the task of unfolding and replacing multiplication with addition: $t * f \rightarrow t + t * (f - 1)$ for $f \geq 2$. For iterative simplification purposes, we quantify both patterns at the level of additions so that the output of one simplification step can be used as the input to the next step: i.e. $e + t * f \rightarrow e' + t * (f - 1)$, where $e' = e + t$. Now, we encode the simplification:

```
fun timesToPlus (Plus(e, Mult(t, Num x)))
      = Some (Plus(Plus(e, t), Mult(t, Num (x - 1))))
  | timesToPlus t
      = None t
val timesToPlus : expr → (expr,expr) options

fun timesToPlus' (Plus(e, Mult(t, Num 2)))
      = Some (Plus(Plus(e, t), t)) (* no more Mult *)
  | timesToPlus' t
      = None t
val timesToPlus' : expr → (expr,expr) options

val timesToPluses = choice timesToPlus timesToPlus';
val timesToPluses : expr → (expr,expr) options
```

For exhaustive replacement of multiplications, we now wrap this strategy in a fixed-point iterator:

```
fun fix s t =
  case (s t) of
    Some t' ⇒ fix s t': ('a,'a) options
  | None _  ⇒ Some t;
val fix : ('a → ('b,'a) options) → 'a → ('a,'a) options

val removeTimes = fix timesToPluses;
val removeTimes : expr → (expr,expr) options
```

However, testing the resulting strategy *removeTimes* on some inputs would reveal that it does not terminate. This is because of the incorrect ordering of strategies in the conditional composition *timesToPluses*. In particular, the input pattern of the first rewrite rule – *Num x*, where *x* is a pattern-match variable – subsumes the input pattern of second rewrite rule – *Num 2*, which is a concrete term – leading to a non-terminating application of the first strategy producing the sequence of reduct terms . . . 3, 2, 1, 0, -1, -2 . . .. In other words, the rewrite rule expressing the base case (*timesToPlus'*) that removes further applicability of iterative simplification never fires. Instead, the inductive case always applies leading to non-termination.

As before, the type system is too coarse and does not raise any flags about the dead code and we encounter the error only during run-time testing. A proper composition is the reverse of the initial attempt:

```
val timesToPluses' = choice timesToPlus' timesToPlus;
```

4. STATIC ANALYSIS

In the previous section, we have discussed common pitfalls of programming rewrite strategies. Due to the inherent nature of such programming, the errors are beyond the reach of the strongly typed frameworks of functional programming. In this section, we propose an approach for improving the precision of the static type analysis.

We begin the discussion from the definition of the syntax and semantics of a core strategic rewriting language *StratCore*. We use an executable specification style of a datatype-based syntax and interpreter-based semantics embedded in SML to avoid potential ambiguities of an abstract notation.

Fig. 2 summarizes the definition of *StratCore*. Terms follow the standard representation of variables and proper terms [1]. The strategic rewriting language consists of the primitive abstractions of rewrite rules, identity and failing rewrites and their sequential and conditional compositions.

The interpreter of the language is a function that implements the semantics of applying a strategy to a term and deriving a new term in case of success or leaving the input term unchanged in case of failure. The definition of the interpreter uses a standard library function *rewrite : term * term → term → term* (p.81, Fig. 4.7 in [1]) that either rewrites an input term or raises a match exception.

To enable detailed type analysis of rewrite strategies, we adopt a fine-grain model of types, where a term's structure itself is the type of the term. This enables static detection of structural incompatibilities between a rule's left-hand side and an input term in an application *Rule(l, r) t*. In addition, we use integer-valued type variables to assign types to term variables. Since rewrite rules are abstractions, whose right-hand side may refer to variables bound by the left-hand side, we use contexts in the form of association lists to track variable type bindings. Fig. 3 summarizes all of these constructs.

The analysis of rewrite strategies involves a preliminary step of assigning types to terms and strategies. Once the types are calculated, we can turn to the actual analysis of applying a strategy to a term.

```
datatype term      = T of string * term list
                   | V of string;
datatype stratcore = Id
                   | Fail
                   | Rule   of term  * term
                   | Seq    of stratcore * stratcore
                   | Choice of stratcore * stratcore;

val rec interpret : stratcore -> (term, term) strategy
  = fn Id             => (fn t => Some t)
     | Fail           => (fn t => None t)
     | Rule (l, r)    => (fn t => Some (R.rewrite (l,r) t)
                              handle _ => None t)
     | Seq (s, s')    => (fn t => case (interpret s t) of
                              Some t' => interpret s' t'
                            | None t' => None t')
     | Choice (s, s') => (fn t => case (interpret s t) of
                              Some t' => Some t'
                            | None t' => interpret s' t')
```

Figure 2: Syntax and semantics of *StratCore*

```
datatype tyT = TyTerm of string * tyT list | TyVar of int;
datatype tyS = TyRule of tyT * tyT;
type context = (string * tyT) list ;
```

Figure 3: Types and contexts

```
(* updates a context by mapping term vars to type vars *)
val rec update: context -> term -> context
  = fn ctx => (fn V x      => if R.indom x ctx then ctx
                              else (x, nextTyVar())::ctx
                | T(_,ts) => foldl (fn (t,ctx') => update ctx' t)
                              ctx ts);

(* queries a typing context by a variable name *)
val rec query: string -> context -> tyT
  = fn x => (fn ((y,t)::ctx) => if x = y then t else query x ctx
              | [ ]          => raise FreeVarError)

(* calculates the type of a term *)
val rec typeOfT: context -> term -> tyT = fn ctx =>
  fn V x          => query x ctx
   | T (f, ts)    => TyTerm(f, map (typeOfT ctx) ts)

(* calculates the type of a strategy *)
and typeOfS: context -> stratcore -> tyS list = fn ctx =>
  fn Id             => let val t = nextTyVar()
                       in  [TyRule (t, t)]
                       end
   | Fail           => [ ]
   | Rule   (l, r) => let val ctx' = update ctx l
                           val tL = typeOfT ctx' l
                           val tR = typeOfT ctx' r
                       in  [TyRule (tL, tR)]
                       end
   | Choice (l, r) => let val sL = typeOfS ctx l
                           val sR = typeOfS ctx r
                       in  case isReachable (sL, sR) of
                             true   => sL @ sR
                           | false  => raise DeadCodeError
                       end
   | Seq    (l, r) => let val sL = typeOfS ctx l
                           val sR = typeOfS ctx r
                       in  case calcPaths (sL, sR) of
                             [ ] => raise FailingStrategyError
                           | ss   => ss
                       end
```

Figure 4: Types of terms and strategies

Fig. 4 summarizes the calculation of types. Here, function *typeOfT* calculates the type of a term as the term itself with term variables replaced by their type bindings in the current context. The type of strategies is computed by function *typeOfS*, which computes the untagged union type [12] of possible rewrites in a given strategy as a list. In particular, strategy *Id* always succeeds and leaves the input term unchanged. This behavior is modeled by a list, whose sole element is the identity rewrite type. Note that fresh type variables are obtained from function *nextTyVar: unit* $\rightarrow$ *tyT*, which increments a reference counter upon return. Strategy *Fail* always fails. This is modeled by an empty list of rule types.

The type of a rewrite rule is a rule type obtained by first updating the current context with the variables of the rule's left-hand side mapping unique term variables to fresh type variables, and then computing the rule's term types based on the resulting context. This enables static detection of free variables in a rule's right-hand side that are not bound by the rule's left-hand side.

Calculation of a type for a conditional composition proceeds by computing the types of the two operand strategies. Conditional composition is well-formed if the right operand is reachable and is not subsumed by the left operand as discussed in Section 3.3. The type of a well-formed conditional composition is a union of the types of its operands. We model the union of two types as concatenation of lists of possible types of each operand strategy.

```
(* checks if the the first type is a subtype of the second *)
val rec subtype: tyT * tyT -> bool
  = fn (_              , TyVar _       )  => true
     | (TyTerm (x,xs) , TyTerm (y,ys) )  => x = y andalso
                                              subtypeL (xs, ys)
     | (x             , y             )  => if x = y then true
                                             else false

(* checks if the first list is pairwise subtype of the second *)
and subtypeL: tyT list * tyT list -> bool
  = fn ([ ]   , [ ]   ) => true
     | ([ ]   , _     ) => false
     | (_     , [ ]   ) => false
     | (x::xs, y::ys) => subtype (x, y) andalso subtypeL (xs, ys)

(* checks if second list's strategies are reachable from first's *)
val rec isReachable: tyS list * tyS list -> bool
  = fn ([ ]                    , _                    ) => true
     | (_                      , [ ]                  ) => true
     | (TyRule (a,b)::xs, TyRule (c,d)::ys) =>
          not (subtype (c,a)) andalso
          isReachable (TyRule (a,b)::[ ], ys) andalso
          isReachable (xs, TyRule (c,d)::ys)
```

Figure 5: Analysis of conditional compositions

The constraints of a well-formed conditional composition are checked by function *isReachable*, whose definition is summarized in Fig. 5. In the first base case of a failing strategy (i.e. *[ ]*) as the left operand, reachability is automatically satisfied. In the second base case of a failing strategy as the right operand, reachability is satisfied in a degenerative sense in that it does not matter whether a failing strategy is reachable. In the more interesting case of two potentially successful rewrites types *TyRule(a, b)* and *TyRule(c, d)*, the second rule is *reachable* if its left-hand side c is not a subtype of the left-hand side a of the first rule. We can recall from Section 3.3 that if the subtype relation holds (i.e. $c <: a$), then (1) if the earlier rule $a \rightarrow b$ fires, then rule $c \rightarrow d$ is not even attempted, and (2) if the earlier rule does not fire, then rule $c \rightarrow d$ will also not fire because c will not match an input term, which did not match a, due to subtyping. Finally, in the inductive case, the first rule $a \rightarrow b$ should not subsume other rules ys in the second strategy and other rules xs in the first strategy should not subsume any of the rules in the second strategy.

The subtyping relation $\cdot <: \cdot$ is defined by the mutually recursive functions *subtype* and *subtypeL*. Intuitively, a term is a subtype of another term if it has the same structure yet it is more specific or more defined than another term. Thus, type variables are maximal types in this relation.

Example.

Below is an example of a static detection of the kinds of errors identified in Section 3.3:

```
(* e + t * x -> e + t + t * (x - 1) *)
val timesToPlus
  = Rule(T("Plus", [V "e",
                    T("Mult", [V "t",
                               T("Num", [V "x"]) ]) ]),
         T("Plus", [T("Plus", [V "e", V "t"]),
                    T("Mult", [V "t",
                               T("Num", [V "x",
                                         T("-", [ ]),
                                         T("1", [ ]) ]) ]) ]) );
(* e + t * 2 -> e + t + t *)
val timesToPlus'
  = Rule(T("Plus", [V "e",
                    T("Mult", [V "t",
                              T("Num", [T("2", [ ])]) ]) ]),
         T("Plus", [T("Plus", [V "e", V "t"]),
                    V "t"]) );

val timesToPluses = Choice (timesToPlus, timesToPlus');

val testChoice = typeOfS [ ] timesToPluses; (* raises DeadCodeError *)
```

□

Returning back to Fig. 4, the type of a sequential composition is computed by calculating types of the operand strategies and passing these onto function *calcPaths*, which computes a pair-wise sequential composition of each type on the left with each type on the right. We can recall from Section 3.2 that a sequential composition is well-formed if all strategies in its sequence can succeed. This constraint is checked by ensuring that the resulting list of possible types of the sequential composition is not empty. Otherwise, the sequential composition will always fail at run-time indicating a programming error.

```
(* calculates paths through sequential composition *)
val rec calcPaths: tyS list * tyS list -> tyS list
  = fn ([ ]                    , _                    ) => [ ]
     | (_                      , [ ]                  ) => [ ]
     | (TyRule(a, b)::xs, TyRule(c, d)::ys) =>
         let val ctx' = (SOME (U.unify (b, c))) handle _ => NONE;
         in  (case ctx' of
                 SOME ctx => [TyRule(U.lift ctx a, U. lift ctx d)]
               | NONE     => [ ]) @
              calcPaths (TyRule (a, b)::[ ], ys) @
              calcPaths (xs, TyRule (c, d)::ys)
         end
```

Figure 6: Analysis of sequential compositions

Fig. 6 summarizes the definition of *calcPaths*. In the base cases of a failing strategy (*[ ]*) on the left or on the right, the resulting list of possible types is empty. Otherwise, a sequential composition of two rewrite rules $a \rightarrow b$ and $c \rightarrow d$

is well-formed if the output term type b of the first rule is unifiable with the input term type c of the second rule. We use standard library functions *unify: ty * ty → context* and *lift: context → ty → ty* (p.80, Fig. 4.5 in [1]) that respectively create and apply (type) substitutions (cf. the paper's accompanying source repository). Finally, in the inductive case, we compose the rule $a \to b$ with the remaining rules on the right ys and compose the remaining rules on the left xs with all of the rules on the right. The concatenation of all resulting lists forms the union type of possible paths through the sequential composition.

Example.
The following illustrates a detection of sequential composition errors identified in Section 3.2:

```
(* 0 * x -> 0 *)
val timesZero
  = Rule(T("Mult", [T("Term", [T("Num", [T("0", [ ]) ])  ]),
                    V "x"]),
         T("Term", [T("Num", [T("0", [ ])  ])  ])  );

(* x * 0 -> 0 *)
val timesZero'
  = Rule(T("Mult", [V "x",
                    T("Num", [T("0", [ ])  ])  ]),
         T("Term", [T("Num", [T("0", [ ])  ])  ])  );

(* 1 * x -> x *)
val timesOne
  = Rule(T("Mult", [T("Term", [T("Num", [T("1", [ ]) ])  ]),
                    V "x"]),
         T("Term", [V "x"])  );

(* x * 1 -> x *)
val timesOne'
  = Rule(T("Mult", [V "x",
                    T("Num", [T("1", [ ])  ])  ]),
         V "x" );

val timesZeroes = Choice (timesZero, timesZero');

val timesOnes = Choice (timesOne, timesOne');

val simpTimes = Seq (timesZeroes, timesOnes);

val testSeq = typeOfS [ ] simpTimes; (* raises FailingStrategyError *)
```

□

```
(* applies a list of rewrites to a term to get a list of output terms*)
val rec apply: tyT -> tyS list -> tyT list = fn tt =>
  fn [ ]                 => [ ]
   | TyRule (l,r ):: ss => let val ctx' = SOME (U.unify (l,tt))
                                          handle _ => NONE
                           in  (case ctx' of
                                  SOME ctx => [U.lift ctx r]
                                | NONE     => [ ])
                                @
                                (apply tt ss)
                           end

(* given a strategy and a term, computes the list of possible types *)
val rec typeCheck: stratcore -> term -> tyT list
  = fn s => fn t => let val ss = typeOfS [ ] s;
                        val tt = typeOfT [ ] t;
                    in  case (apply tt ss) of
                          [ ] => raise ApplicationError
                        | tts => tts
                    end
```

Figure 7: Analysis of strategic applications

Having defined the functions that compute types of terms and strategies, we are now ready to turn to the main subject of type analysis: strategy application. Fig. 7 summarizes the definition of function *typeCheck* that performs the analysis. Application of a strategy to a term is well-formed if the list of possible types resulting from reducing the strategy type *ss* on the term type *tt* is not empty. Otherwise, a strategic type error occurs because the strategy is not defined for the term.

Example.
The following illustrates detection of application errors identified in Section 3.1:

```
val r1 = Rule(T("Expr", [V "x",
                         T("+", [  ]),
                         T("0", [  ])  ]),
              V "x" );
val testApp1
  = typeCheck r1 (T("Expr", [T("Expr", [T("a", [ ])  ]),
                                        T("+", [  ]),
                                        T("0", [  ])  ])  );
    (* = [TyTerm ("Expr", [TyTerm ("a", [ ]) ])]  *)
val testApp2
  = typeCheck r1 (T("Expr", [T("Expr", [T("a", [ ])  ]),
                             T("+", [  ]),
                             T("1", [  ])  ])  ); (* raises ApplicationError*)
```

□

5. DISCUSSION OF TYPING PROPERTIES

In the previous section, we have presented the type analysis of rewrite strategies. In this section, we discuss its properties: in particular the soundness of the analysis. A type system is sound if well-typed terms do not go wrong. In the strategic rewriting setting, going wrong means applying a top-level strategy to a term, for which it is undefined: i.e. *interpret s t = None t*. Note that constituent strategy failures may be handled by the composite strategy and thus applications of the form *Choice(fail, id) t* are well-typed with the type *T::[ ]* (i.e. a non-empty list). A strategy goes wrong when the failure always bubbles up and escapes the strategic controls unhandled.

THEOREM 1. *If* $\vdash s\ t : (T :: Ts)$*, then interpret* $s\ t =$ *Some* t' *such that* $\vdash t' : T'$ *and* $T' \in (T :: Ts)$*.*

Proof By induction on the structure of s. At each step of the induction, we assume that the theorem holds for sub-strategies (if any) of s:

Identity If s is *Id*, then $\vdash s\ t : [T]$ with $\vdash t : T$. Since *interpret Id t = Some t*, it is clear that $T \in [T]$.

Failure If s is *Fail*, then $\vdash s\ t : [\ ]$, the condition of the theorem about a well-typed application does not hold and the theorem is vacuously satisfied.

Rewrite rule If s is *Rule (l, r)*, then we have two subcases under standard matching and unification semantics [1], which is used by both interpretation and type-checking:

Rewrite succeeds If the rewrite succeeds, we obtain $\vdash s\ t : [R']$ and *interpret (Rule (l, r)) t = Some r'* with $\vdash r' : R'$. Thus, we have $R' \in [R']$.

Rewrite fails If the rewrite fails, we obtain $\vdash s\ t : [\ ]$ and *interpret (Rule (l, r)) t = None t*. The theorem's condition about a well-typed application does not apply.

Sequence If s is *Seq* $(s_1,\ s_2)$ and $\vdash s\ t : (T :: Ts)$, we know that there exists at least one valid rewrite path through s based on Figures 4 and 6. Further, term t matches an input pattern of at least one rewrite path, because $(T :: Ts)$ is non-empty. Since the semantics also uses standard matching, *interpret* $s_1\ t$ = *Some* t' and *interpret* $s_2\ t'$= *Some* t'' such that $t'' : T''$ and $T'' \in [T :: Ts]$.

Choice If s is *Choice* $(s_1,\ s_2)$, then we use the induction hypothesis and assume that the theorem holds for constituents s_1 and s_2. Since the types of $s_1\ t$ and $s_2\ t$ are concatenated (Figure 4), we know that the actual type T' is within the list of possible types.

□

6. RELATED WORK

In [10], the importance of error detection in strategic rewriting has been previously highlighted as a major research problem. The present paper makes relevant contributions in so far that specific categories of programming errors are identified, and a corresponding static analysis is devised.

In [8], a relatively conservative type system for rewrite strategies, including traversal strategies, was developed. More specifically, system S—the core of program transformation system Stratego [15]—is extended with new syntax and semantics to support types. Our contributions advance this work by improving the expressivity of type analysis. Most notably, instead of using *sorts* to assign types to rewrite rules, we incorporate *constructors* into types. This improves the analytical precision because application of a rule, which acts on one constructor of a sort, to a term derived from a different constructor of the same sort, while being well-typed under the previous approach, will always fail, which is statically detected under the current approach.

An extended strategic rewriting core language that includes one-layer traversals over a single-sorted term language of natural numbers has been recently formalized in an Isabelle/HOL-based model [7]. There, success and failure behavior of strategies has been analyzed from the perspective of *infallibility*: i.e., does a given strategy always succeed? Our analysis takes a dual view on this issue from the perspective of successfulness: i.e., can a given strategy succeed? Both approaches approximate correctness: in the first case by flagging over-specified strategies, and in the second case by flagging under-specified strategies.

Among the related work on transformations of tree- structured data are the W3C standards on XPath, XQuery and XSLT [16]. In particular, an XSLT style-sheet uses XPath and/or XQuery expressions to select elements within an XML document and uses templates to transform the elements. In other words, selection criteria could be viewed as pattern matches and templates as rewrite rules among many other cross-domain similarities [9, 3]. In this domain, an important type-checking question is whether the result of an XSLT transformation conforms to an intended type. This is due to the asymmetry arising from parser-based validation on inputs, but none on outputs. To cope with this problem, regular expression types [5] have been proposed to validate XML transformations [6] along with efficient type-checking implementations [4]. Other related type-checking questions are whether selection criteria return an empty set of nodes leading to a template that can never fire and whether one template is subsumed by another.

7. CONCLUSION

In this paper, we have presented a type analysis of rewrite strategies for the purposes of automatic detection of strategies that always fail: in particular incorrect strategy compositions and incorrect strategy applications. The type system closely approximates the run-time behavior by using fine-grain constructor-based types and models strategic outcomes using union types. As a result, the system can detect a larger than previously possible number of errors. This can lead to a substantial reduction in testing and debugging of strategies in large and complex problems.

Future work in this research includes expansion of the scope of the analysis to include one-layer traversals, which combined with recursion enable full term traversal and rewriting. The type system presented here can be extended to handle recursive closure by abstracting the types along at least two dimensions: (a) prune a type's tree structure toward its root and (b) combine multiple constructor alternatives derived from the same parent symbol into one parent symbol (e.g. *[Int(...), Float(...)]* into *[Number(TyVar(i))]*).

8. REFERENCES

[1] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[2] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.

[3] A. Cunha and J. Visser. Transformation of structure-shy programs: applied to XPath queries and strategic functions. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'07)*, pages 11–20. ACM Press, 2007.

[4] J. N. Foster, B. C. Pierce, and A. Schmitt. A Logic Your Typechecker Can Count On: Unordered Tree Types in Practice. In *Proceedings of the ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X)*, pages 80–90, 2007.

[5] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 2nd edition edition, 2000.

[6] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2005.

[7] M. Kaiser and R. Lämmel. An Isabelle/HOL-based Model of Stratego-like Traversal Strategies. In *Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'09)*, September 2009.

[8] R. Lämmel. Typed Generic Traversal with Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54:1–64, 2003.

[9] R. Lämmel. Scrap your boilerplate with XPath-like combinators. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages (POPL'07), pages 137–142. ACM Press, 2007.

[10] R. Lämmel, S. Thompson, and M. Kaiser. Programming errors in traversal programs over structured data. *ENTCS*, 238(5):135–153, 2008.

[11] R. Lämmel and J. Visser. A Strafunski Application Letter. In *PADL'03: Proceedings of Practical Aspects of Declarative Programming*, volume 2562 of *LNCS*, pages 357–375. Springer, Jan. 2003.

[12] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] J. D. Ullman. *Elements of ML Programming*. Prentice Hall, Upper Saddle River, NJ, USA, 1998.

[14] M. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Transactions Software Engineering Methodology*, 12(2):152–190, 2003.

[15] E. Visser, Z. e. A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[16] W3C. W3C XML Transformation Standards: c2010. Available from: http://www.w3.org/standards/xml/transformation.

[17] V. Winter and M. Subramaniam. The Transient Combinator, Higher-order Strategies, and the Distributed Data Problem. *Science of Computer Programming*, 52:165–212, 2004.