The Whiley Rewrite Language (WyRL)

David J. Pearce
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
djp@ecs.vuw.ac.nz

Abstract

The Whiley Rewrite Language (WyRL) is a standalone tool providing a domain-specific declarative rewrite language and code generator. The tool is currently used to generate a critical component of the Whiley verifying compiler, namely the automated theorem prover. The tool automatically generates Java source code from a given rule set. The runtime library provides support for different heuristics to control aspects of the generated system, such as the order in which rewrite rules are applied. Novel aspects of WyRL include support for true union types and the ability to work with cyclic terms.

Categories and Subject Descriptors I.2.3 [*Deduction and Theorem Proving*]: Inference engines

General Terms Languages, Theory, Verification

Keywords Term Rewriting, Deductive Theorem Proving

1. Introduction

General purpose term rewrite systems have been used in the engineering of programming languages and related tools for some time [1]. Success has been achieved in a wide range of areas, such as: *program transformation* [2, 3], *program analysis* [4], *formal verification* [5, 6], and *domain-specific languages* [7]. Numerous industrial-strength rewriting tools have also been developed, with notable examples including: CafeOBJ [5, 8], ELAN [9], Maude [10, 11], Stratego [2], ASF+SDF [7, 12] and Rascal [4] (amongst others). To that end, there is little doubt that certain problems can benefit significantly from being expressed with rewrite rules. Specifically, the separation of rewrite rules from the application *strategy* allows easy experimentation with different strategies and, furthermore, conceptually helps to divide up the problem. For example, in the development of theorem proving tools, the benefits of separating rewrite rules from their (often complex) application strategies are well-known [1]. Despite this, many widely-used theorem proving tools (e.g. [13–16]) are still implemented in an ad-hoc fashion, making them difficult to debug and maintain.

In this paper, we introduce the *Whiley Rewrite Language (WyRL)* which is a standalone tool providing a domain-specific declarative rewrite language and code generator. The tool automatically generates Java source code from a given declarative rule set to ensure the generated rewrite engine is efficient. This rewrite engine repeatedly applies rewrite rules according to user-supplied heuristics until no further applications are possible. Rewrite rules are divided into two categories: *reductions* and *inferences*. The intuition is that the former reduce the "size" of the term being rewritten, whilst the latter may increase it. An intricate mechanism is used to ensure this process terminates (more on this later).

The WyRL tool was developed specifically to aid the construction of an automated theorem prover, called WyCS, which forms a critical component of the Whiley verifying compiler [17]. Whiley is a programming language designed from scratch to simplify the process of software verification. Its verifying compiler statically checks all function specifications and data-type invariants hold. More information about Whiley can be found elsewhere [17, 18].

Although WyRL was originally designed specifically for use with the Whiley compiler, it has many applications outside this. For example, one can easily develop the core component of a type checker for a wide range of type systems, particularly those involving recursive types. Indeed, the tool is used to generate the key components of its own type checker, and will eventually be used to replace the hand-coded system for normalising and checking recursive types for the Whiley language itself. Similarly, the tool can be used to develop deductive verification systems, such as SMT solvers (e.g. [13, 15]), with relative ease.

Compared with previous tools, the main novelty of WyRL lies in the following:

1. **Language expressivity.** The rewrite language used by WyRL contains a number of interesting features, including elegant support for both ordered and orderless pattern matching and support for true union types (i.e. which are non-disjoint, as opposed to sum types).
2. **Term Graphs**. Unlike the majority of existing tools, WyRL operates on general graphs which may be cyclic (rather than just e.g. trees or DAGs). For example, identical subterms are represented exactly once. Likewise,

the ability to rewrite cyclic structures means that certain rewrite problems can be expressed within WyRL, such as that of normalising recursive types.

3. **Code Generation**. WyRL compiles rewrite rules into Java source code so they can be executed directly on the JVM. The outer loop controlling the application of rules is provided as a library function which operates over an arbitrary rule set and application strategy.

WyRL is under an open source license and is available from `github.com/Whiley/WhileyRewriteLanguage`.

2. Language Overview

We now give a brief overview of WyRL. In the following section, we examine the implementation in more detail.

2.1 Basics

The fundamental building blocks are *terms*. For example:

```
term True
term False
define Bool as True | False // Booleans

term Var(string) // Variables
term Not(BExpr) // Negations

define BExpr as Bool | Var | Not(BExpr)
```

Here, we defined a simple language for describing boolean expressions of the form `True`, `False`, `Var("X")`, `Not(True)`, `Not(Not(False))`, `Not(Var("Y"))`, etc. Rewrite rules can employ pattern matching over this language as follows:

```
reduce Not(Bool b):
    => False, if b == True
    => True

reduce Not(Not(BExpr e)):
    => e
```

These rewrites implement two obvious simplifications for expressions in our language. Each rewrite rule consists of a pattern and one or more *cases* delineated by "=>". The first rule matches either `Not(True)` or `Not(False)` and employs a *conditional case* to distinguish them. Cases are tried in order of occurence and, hence, the second case of this rule is applied only if the first is not. The second rule matches terms such as `Not(Not(Var("X")))`, `Not(Not(Not(False)))`, etc.

The order of rule applications is unspecified and there are two valid applications of this rule to `Not(Not(Not(False)))` (i.e. where `e` either binds to `False` or `Not(False)`) and we cannot determine which will be applied.

2.2 Pattern Matching over Collections

Pattern matching is one of the fundamental building blocks underlying WyRL. In the rewrites above, the patterns were simple and the ordering of subterms not a consideration. WyRL supports both *ordered* and *unordered* collections of subterms, where the former correspond roughly to lists or arrays and the latter to sets or bags.

Unordered Collections. The following illustrates a compound term with an unordered set of subterms:

```
term And{BExpr...} // Logical conjunction
```

Here, the "`...`" in "`BExpr...`" indicates *zero-or-more* occurrences. Since sets are unordered, `And{True,False}` is indistinguishable from `And{False,True}`. Rewrites over sets employ unordered (i.e. commutative) pattern matching:

```
reduce And{Bool b, BExpr... xs}:
    => False, if b == False
    => True, if |xs| == 0
    => And (xs)
```

The above matches any instance of `And` with *at least one subterm* and where *that subterm is an instance of* `Bool`. Thus, it will match `And{True}` and `And{Not(False),True}`, but not `And{Not(False)}` or `And{}`. Furthermore, there are two possible applications of this rule to the term `And{True,False}` and the order in which they will be applied is unspecified.

The following gives another illustration. This will, for example, reduce `And{Not(Var("X")),Var("X")}` to `False`:

```
reduce And{Not(BExpr x), BExpr y, BExpr... ys}:
    => False, if x == y
```

Here, the rewrite matches instances of `And` with *at least two subterms* and where *one is a negation and the other a general instance of* `BExpr`. Finally, we note that multisets (or bags) are also supported. For example, `Sum{|Expr...|}` is a term with zero or more subterms which need not be distinct.

Ordered Collections. WyRL includes a *list* type which provides *ordered* pattern matching. The following illustrates (where we assume `BExpr` is extended to include `Eq`):

```
term Num(int)          // Integer numbers
term List[Expr...]     // List constructors
define Expr as Var | Num | Bool | List
term Eq{|Expr, Expr|} // Equalities

reduce Eq{|List[Expr... xs],List[Expr... ys]|}:
   => False, if |xs| != |ys|
   => let r = {Eq[xs[i],ys[i]] | i in 0 .. |xs|}
      in And(r)
```

This rule reduces equality of lists to that of ensuring element equality (and, indeed, matching numbers of elements).

2.3 Comprehensions

Manipulating collections in WyRL is achieved with the use of comprehensions, as the following illustrates:

```
term Or{BExpr...} // Logical disjunctions

reduce Not(And{BExpr... xs}): // Apply DeMorgan's Law
    => let ys = { Not(x) | x in xs }
       in Or(ys)
```

As expected, this rule distributes negations over conjunctions to produce disjunctions of negations.

```
term LessThan[Expr,Expr] // Strict inequalities

reduce LessThan[Num(int x), Num(int y)]:
    => True, if x < y
    => False

infer And{LessThan[Expr e1, Expr e2] l1,
          LessThan[Expr e3, Expr e4] l2,
          BExpr... bs}:
    => let rs = {l1,l2,LessThan[e1,e4]}
       in And (bs ++ rs), if e2 == e3
```

Figure 1. Illustrating transitive closure over inequalities.

2.4 Inferences versus Reductions

The rewrites we have seen thus far have all been *reductions*. The intuition is that reductions reduce the term being rewritten in some way. For example, the rules shown in §2.1 always reduce the overall number of terms. In contrast, the reduction rule in §2.3 may *increase* the number of terms; nevertheless, it is still reducing the overall number of `And` terms and a proof of termination could be based around this.

The WyRL tool does not guarantee termination and it is the programmer's responsibility to ensure this. Nevertheless, in developing an automated theorem prover for Whiley, we have encountered situations which, without care, clearly will not terminate. Figure 1 illustrates such an example where an *inference rule* implements transitive closure of inequalities. Now, consider consider this sequence of rule applications:

$$\begin{array}{llr} 1 < x \wedge x < 2 & \Longrightarrow 1 < x \wedge x < 2 \wedge 1 < 2 & \textit{(infer)} \\ & \Longrightarrow 1 < x \wedge x < 2 \wedge \textit{true} & \textit{(reduce)} \\ & \Longrightarrow 1 < x \wedge x < 2 & \textit{(reduce)} \end{array}$$

Here, the initial state is reached again after application of the inference and reduction rules and, without care, this would be continue *ad infinitum*. The distinction between inference and reduction rules provides one mechanism for avoiding this. Specifically, the application of an inference rule is only considered *successful* if it introduced new information. Hence, after applying an inference rule, the term is maximally reduced and, if the result is equivalent to that beforehand, the inference is *unsuccessful*. Hence, the above sequence doesn't apply as the inference rule is unsuccessful.

2.5 Union Types

WyRL supports *union types* which are non-disjoint. First, we note that defined types are really just macros. Thus, `Bool` as declared in §2.1 is not actually a type; rather, it is simply a name referring to the *underlying* type `True|False`. To illustrate, let us extend our running example:

```
term Eq{|Expr,Expr|}          // Equalities
define Value as Num | Bool   // Values

reduce And{Eq{|Var v,Value n|}, BExpr... xs}:
   => let ys = { x[v\n] | x in xs }
      in And([Eq{|v,n|}] ++ ys)
```

Here, `x[v\n]` represents `x` with all occurrences of `v` replaced by `n`. This rule substitutes a variable which equals a known value throughout a conjunction. `Value` is implicitly a subtype of `Expr` as all terms in the former are fully contained in the latter. In a system based on sum types, this is not possible because `Expr` and `Value` would be *disjoint*.

2.6 Cyclic Terms

A key feature of WyRL is the ability to handle *cyclic terms* (although the examples thus far don't require this). The presence of recursive types in Whiley and in WyRL itself warrant this feature (see e.g. Pierce for more on recursive types [19]). To illustrate, let us consider the following (significantly cut-down) encoding of WyRL types in WyRL itself:

```
term String                              // string types
term Term[string,Type...]                // term descriptors
term Or{Type...}                         // union types
define Type as String | Term | Or // types (simplified)

reduce Or{Or{Type... xs}, Type... ys}:
    => Or (xs ++ ys)
```

Now, the type `Var(string)` from our example in §2.1 is encoded internally as `Term["Var",String]`. Encoding WyRL types within itself allows us to leverage the rewrite system (e.g. for handling union types). For example, the type `Not(Bool|(Bool|Var(string))` is represented internally as:

```
Term["Not",Or{
  Term["Bool"],Or{
    Term["Bool",Term["Var",String]
} }]
```

The reduction rule given above matches and reduces this type to `Not(Bool|Var(string))`.

We now consider how the type `BExpr` from §2.1 is encoded internally. The challenge is that `BExpr` is a recursive type. We can expand `BExpr` to its underlying type, written as μ`X.(True|False|Var(string)|Not(X))`. Here, μ`X` is not part of the type itself but merely a notation for expressing in finite form an infinite structure [19]. As shown in the following section, terms in WyRL are represented using a directed graph structure called an *automaton*. Thus, encoding μ`X.(True|False|Var(string)|Not(X))` within WyRL is achieved using a *cyclic* automaton, where μ`X` is the head of the cycle and other occurrences of `X` to back edges.

3. Implementation

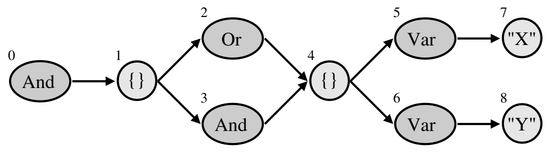
A critical aspect underlying any rewrite system is the manner in which terms are represented [20]. WyRL represents terms internally as a directed graph. This structure contains all terms being rewritten and is referred to as an *automaton*. As expected, our automata can be subjected to the usual range of algorithms, including *minimisation*, *canonicalisation* and *language inclusion* [21]. Due to a lack of space, we omit a formal definition of our automata here. An important invariant maintained over our automata is the following:

DEFINITION 1 (Automata Invariant). *No two equivalent but distinct states exist.*

This invariant has important consequences. For example, equality testing of terms is constant time (i.e. by comparing state identifiers rather than recursively traversing both terms). But, care must be taken when rewriting to maintain this invariant which incurs additional cost.

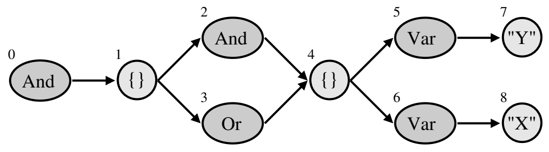
3.1 Automata

Let us first consider the automaton representing the term `And{Or{Var("X"),Var("Y")},And{Var("X"),Var("Y")}}`:



Each state has a unique integer identifier, and their significance is discussed below. Furthermore, we can see how the graph representation enables maximal sharing of states.

Since automata correspond to directed (coloured) graphs, the question of *equivalence* is important as automata may be isomorphic (i.e. equivalent but distinct). For example, the following automaton is isomorphic to that above:



Here, for example, state 2 is an instance of `Or` in the first automaton, but an instance of `And` in the second. Determining equivalence between automata reduces to the graph isomorphism problem, whose computational complexity is neither known to be NP-complete nor in P [22]. Although relatively efficient algorithms exist in practice, these remain too costly for our purposes here and, in fact, our rewriting system avoids them (see below) [23].

3.2 Minimisation

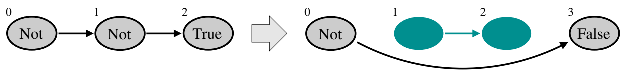
The process of establishing our Automata Invariant is called *minimisation*. Automata constructed externally to represent terms (e.g. by client code such as a parser, etc) must first be minimised. The $O(n^3)$ DFA-minimisation algorithm of Hopcroft and Ullman is used [21]. This operates by initially assuming all states are equivalent. Then, equivalences between states which are immediately non-equivalent are dropped. Following on from this, equivalences are dropped for those whose subterms are not equivalent and this continues until a fixpoint is reached. At this point, any remaining equivalences are considered true equivalences and such states are carefully collapsed together.

Figure 2 illustrates the minimisation algorithm operating on an example automaton. Each matrix represents the equivalence relation at one step in the algorithm, where a gray box marks two states as equivalent. Initially, all states are considered equivalent. After the first step, state 0 is no longer considered equivalent with anything else as no other `And` terms exist. However, state 3 is considered equivalent to 4 because, at that moment, states 6 and 7 are considered equivalent. After the final step, states 2 and 3 are no longer considered equivalent to state 4 as their children are no longer equivalent. At this point, equivalent pairs (i.e. states 2+3 and 5+6) would be collapsed, leaving state 1 with only two subterms.

3.3 Rewriting

Rewriting consists of two parts: *graph rewriting* and *minimisation*. A critical aspect, for efficiency, is an incremental algorithm for maintaining the automaton as minimised during rewriting which reduces the number of calls required to the minimisation algorithm discussed above. Furthermore, it eliminates the need for an automata equivalence check after an inference rule is applied and the automaton reduced.

To better understand rewriting, consider an application of the rule from §2.1 for reducing terms of the form `Not(Bool)`:

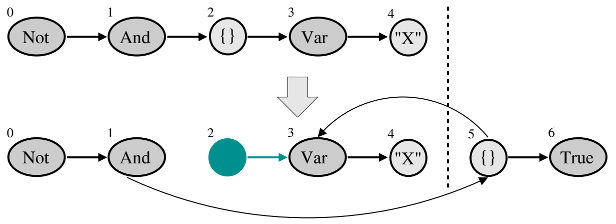


Here, the application created a new state to represent `False` and rewrote state 1 to state 3 by mapping $\{1 \mapsto 3\}$ over all state identifiers. Two *unreachable* states are left which will eventually be eliminated during *compaction*.

To illustrate how automata equivalence checks are avoided when applying inference rule, we use the following rule:

```
infer And{Var(string) var}:
   => And{var,True}
```

Applications of this rule immediately cause a reduction to eliminate `True` (recall §2.2). Hence, the result of applying our inference rule above to `Not(And{Var("X")})` is:



Here, two new states have been created to represent the subterm `{Var("X"),True}`, where `Var("X")` is reused from before. The dashed line is referred to as the *pivot* — namely, the point above which all new states are created. In any subsequent reductions after an inference, no reachable states can remain above the pivot if the resulting automaton is to be equivalent to the original. Unreachable states are not deleted during this process but are retained, while new states are only created when no existing states match. Thus, reducing the above term `And{Var("X"),True}` to `And{Var("X")}` returns us to *exactly* the same configuration we began from (modulo some unreachable states above the pivot).

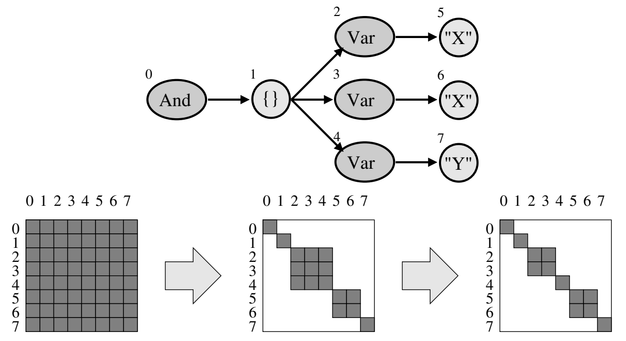


Figure 2. Illustrating minimisation of a small automaton.

```
class Reduction_1 implements ReductionRule {
 public void probe(Automaton aut,
       int root, List<Activation> acts) {
   int r0 = root;
   Automaton.State s0 = aut.get(r0);
   if(s0.kind == K_Not) {
     Automaton.Term t0 = (Automaton.Term) s0;
     int r1 = t0.contents;
     Automaton.State s1 = aut.get(r1);
     if(s1.kind == K_Not) {
       Automaton.Term t1 = (Automaton.Term) s1;
       int r2 = t1.contents;
       int[] binding = new int[]{r0, r1, r2};
       acts.add(new Activation(this, binding));
  } } }
  public int apply(Automaton aut, int[] binding){
    int r0 = binding[0];
    int r2 = binding[2];
    if(r0 != r2) { return aut.rewrite(r0,r2); }
    else { return Automaton.K_VOID; }
} }
```

Figure 3. Generated code for a rule reducing negations.

3.4 Compilation

Each rule is compiled into Java source code as an instance of `RewriteRule`. This interface contains two functions: `probe()` and `apply()`. The former determines whether a rule can be applied to a given automaton state, whilst the latter actually applies it to transform the automaton. Keeping these functions separate allows the system to probe rules over different states before deciding which to apply. This is important as the order in which rules are applied can, as expected, have a significant effect on overall performance.

Figure 3 illustrates the Java code generated for the rule from §2.1 for reducing `Not(Not(BExpr))` terms. Here, `probe()` accepts a `root` state and a list onto which potential rule activations are placed. A list is used as multiple activations per rule are possible. Each activation retains the binding constructed during `probe()` to avoid reconstructing it in `apply()`. Finally, `rewrite()` replaces all occurrences of the first state with the second in the automaton.

3.5 Heuristics

WyRL supports different heuristics for rule selection which can dramatically affect performance. Users can provide their own heuristics, though this requires an in-depth understanding the system. Thus far, the strategies we have explored are relatively simple, but we intend to consider more in the future. For example, a *randomised* strategy which picks rules at random is easy to implement (though not necessarily very good). In general, rules which will likely reduce overall work should be prioritised. Consider `Not(Not(And{Var("X",False}))`. In this case, several rules are applicable: (i) reducing `Not(Not(BExpr))` (recall §2.1); (ii) reducing `And{False,...}` to `False` (recall §2.2); and (iii) applying DeMorgan's law (recall §2.3). Clearly, in this case, applying DeMorgan's law is not optimal as all work done for this is wasted (i.e. because it can be immediately reduced to `False` regardless).

Our main lines of investigation have been exploring *fair* and *unfair* strategies. A fair strategy ensures all rules are applied equally, and prevents starvation of rules. An unfair strategy, however, might simply choose the first available rule leading to some rules being applied more frequently.

3.6 Native Functions

In developing an automated theorem prover, it was apparent that WyRL's declarative language is insufficient. Thus, a mechanism is included for calling native Java methods from rewrite rules. This requires considerable knowledge of WyRL to do correctly, but is a critical feature as the need to express certain computations is paramount to the primary goal (i.e. that of generating an automated theorem prover).

To illustrate native functions, we consider a cut-down example from the automated theorem prover used in Whiley:

```
infer And{Eq{|Var,Var|} eq, BExpr... bs}:
    => let x = max(eq), // must be greatest
           y = min(eq), // must be least
           cs = { b[x\y] | b in bs }
       in And (eq ++ cs), if x != y

function min(Eq{|Var,Var|}) => Var
function max(Eq{|Var,Var|}) => Var
```

The purpose of this rule is to unify variables known to be equal (often referred to as congruence closure [24]). For example, it reduces the formula $x = y \wedge x \wedge \neg y$ to $x = y \wedge x \wedge \neg x$ which then reduces further. The problem is that, unrestricted, this leads to an infinite loop. For example, the formula $x = y \wedge y = x$ can reduce to $x = y \wedge x = x$, then $x = y \wedge y = y$, then back to $x = y \wedge x = x$ and so on. To prevent this, we desire a lexicographic ordering of variables. That is, if x were considered below y then we would only substitute x for y, but not the other way around.

In the above example, the native functions `min()` and `max()` serve the purpose of implementing an ordering of variables. That is, they respectively identify the smaller and larger of two variables. In the language of WyRL, there is currently no other way to express this requirement. These functions are implemented directly in Java and operate over the underlying representation of terms.

4. Related Work

An early and influential tool is ASF+SDF [7]. This extends the Syntax Definition Formalism (SDF) with a term rewriting capability which supports both conditional and unconditional rewrites. Rules are either interpreted or compiled to native C. An interesting aspect is the strong connection between the context-free grammar of a language and its parse trees (where the latter form the terms used for rewriting). ASF+SDF is naturally suited to the development of domain-specific languages, source code analysis and source code transformations and has found application in industry, particularly to support the prototyping and development of domain-specific languages [12]. Stratego is another tool with a strong focus on program transformation [2]. A key feature is the ability to represent rewrite *strategies* within the language itself through reflection — that is, where rewrite rules and their applications are first-class entities. Thus, one can easily experiment with different approaches.

CafeOBJ is an industrial-strength specification language targeted for use in formal verification [5, 8]. CafeOBJ allows one to describe and manipulate abstract machines and data types and supports both equational and behavioural reasoning. Here, equations are regarded simply as left-to-right rewrite rules to be applied exhaustively. Support for *observational transition systems* makes CafeOBJ suited to modelling and reasoning about event-based systems, such as verifying safety properties for security protocols [6]. ELAN provides a framework for developing deduction systems based on term rewriting, such as theorem provers, logic programming languages, constraint solvers, decision procedures, etc [9]. Like Stratego, ELAN enables sophisticated user-defined strategies to control rewriting by making rules and strategies first-class entities. ELAN is perhaps unusual in having a strong focus on search, rather than deterministic computation [1]. This stems from its goal of supporting the development of deduction systems, which typically perform an exhaustive search for *unsatisfiability* (i.e. contradiction).

Another system with a strong focus on formal verification of models is Maude [10]. Again, Maude supports user-defined rewrite strategies through reflection. An unusual feature is the inclusion of an LTL model checker for checking reachability and invariants. Another interesting aspect of the Maude ecosystem is the number of tools which provide additional services, such as termination checking [25], inductive theorem proving [26], debugging [27], etc.

Rascal is a more recent tool which was strongly influenced by ASF+SDF [4]. Like ASF+SDF, Rascal focuses on program analysis and transformation and essentially extends ASF+SDF with additional pattern matching primitives and features. For example, Rascal does not require concrete syntax be given. Rascal also supports imperative language features, such as I/O and side-effects and, furthermore, during backtracking side-effects are undone (though I/O obviously cannot be).

5. Conclusion

In this paper, we have presented the Whiley Rewrite Language which provides a domain-specific declarative rewrite language and accompanying code generator. Novel features of WyRL include support for true unions and the ability to operate on cyclic terms.

Acknowledgements. The author thanks Lindsay Groves for some insightful feedback on earlier drafts.

References

[1] José Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 81(7-8):721–781, 2012.

[2] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proc. RTA*, pages 357–362, 2001.

[3] M. Bravenboer, A. van Dam, K. Olmos, , and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):1–56, Winter 2006.

[4] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Proc. SCAM*, pages 168–177. IEEE, 2009.

[5] Razvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ. *TCS*, 285(2):289–318, 2002.

[6] Kazuhiro Ogata and Kokichi Futatsugi. Rewriting-based verification of authentication protocols. *ENTCS*, 71:208–222, 2004.

[7] Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach*. AMAST. 1996.

[8] R. Diaconescu and K. Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. AMAST. 1998.

[9] P. Borovanský, C. Kirchner, H. Kirchner, and P. Moreau. ELAN from a rewriting logic point of view. *TCS*, 285(2):155–185, 2002.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *TCS*, 285(2):187–243, 2002.

[11] M. Clavel, F. Durn, S. Eker, P. Lincoln, J. Meseguer, and C. Talcott. The Maude 2.0 system. In *Proc. RTA*, pages 76–87, 2003.

[12] M. Brand, A. Deursen, P. Klint, S. Klusener, and E. Meulen. Industrial applications of ASF+SDF. In *Proc. AMAST*, pages 9–18, 1996.

[13] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *JACM*, 52(3):365–473, 2005.

[14] C. Barrett and C. Tinelli. CVC3. In *Proc. CAV*, pages 298–302, 2007.

[15] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.

[16] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. TACAS*, pages 174–177, 2009.

[17] D. J. Pearce and L. Groves. Whiley: a platform for research in software verification. In *Proc. SLE*, pages 238–248, 2013.

[18] D. J. Pearce and Lindsay Groves. Reflections on verifying software with Whiley. In *Proc. FTSCS*, pages 142–159, 2013.

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

[20] M. Brand, H. Jong, P. Klint, and P. Olivier. Efficient annotated terms. *SPE*, 30(3):259–291, 2000.

[21] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory Languages and Computation*. Addison-Wesley, 1979.

[22] J. Kobler, U. Schoning, and J. Toran. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhauser, Boston, 1993.

[23] B. McKay. Practical graph isomorphism. In *NMC*, pages 45–87, 1981.

[24] Nelson and Oppen. Fast decision procedures based on congruence closure. *JACM*, 27, 1980.

[25] F. Durán and J. Meseguer. On the church-rosser and coherence properties of conditional order-sorted rewrite theories. *JLAP*, 81(7-8):816–850, 2012.

[26] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *JUCS*, 12(11):1618–1650, 2006.

[27] A. Riesco, A. Verdejo, and N. Martí-Oliet. A complete declarative debugger for Maude. In *Proc. AMAST*, pages 216–225, 2010.