Software ENgineering

Rewriting-based languages and systems

J. Heering, P. Klint

REPORT SEN-E0310 DECEMBER 8, 2003

ISSN 1386-369X

Rewriting-based languages and systems

ABSTRACT
Many systems are at least partly or implicitly based on term rewriting. Examples are functional languages, computer algebra systems, and theorem provers. We give a brief survey of these systems, and provide a more in-depth comparison of the features of systems that use term rewriting as their primary execution mechanism. We give links to rewrite tools and projects as well as pointers to notions and techniques covered in the preceding chapters.

1998 ACM Computing Classification System: D.3.4; I.1.3; I.2.3
Keywords and Phrases: rewriting-based language; rewriting-based system
Note: Published as Chapter 15 in: Terese, Term Rewriting Systems, Cambridge University Press, March 2003.

Contents

Chapter 15

Rewriting-based languages and systems

Many systems are at least partly or implicitly based on term rewriting. Examples are functional languages, computer algebra systems, and theorem provers. We give a brief survey of these systems, and provide a more in-depth comparison of the features of systems that use term rewriting as their primary execution mechanism. We give links to rewrite tools and projects as well as pointers to notions and techniques covered in the preceding chapters.
Authors: J. Heering and P. Klint

15.1. Introduction

Ideas and techniques from term rewriting are used in many programming languages and software systems. In some cases they are hardly recognizable as such, for instance, in the algebraic simplifications performed by optimizing compilers. In other cases term rewriting techniques are manifest, as in computer algebra and program transformation systems.

In this chapter we present a brief survey of selected, rewriting-based, languages and systems. A survey cannot, due to lack of space, do justice to the merits of all languages and systems being described. To compensate for this, we provide references to publications or web sites with further detailed information. A survey cannot be complete either: again due to lack of space or ignorance of the authors. To obtain a well-balanced survey we used the following guiding principles. A language or system is included if it satisfies several of the following criteria:

– It is in use and has serious applications. We have used the existence of

[1]To appear as Chapter 15 in the Terese Group's upcoming book *Term Rewriting Systems*, Cambridge University Press, 2002. J. Heering and P. Klint are with CWI, Department of Software Engineering, Amsterdam, The Netherlands. Email: `Jan.Heering,Paul.Klint`@cwi.nl.

a web site as evidence.

- The underlying theory is covered elsewhere in this book.
- It is adequately documented.

Our main grounds for *not* including a language or system have been:

- It is no longer in use. Systems that do not have a web page have probably not been used for years and were not included, although they may be mentioned for historical reasons.
- It is already very well described elsewhere. This is, for instance, the case for functional languages and computer algebra systems.

This chapter is organized as follows. Section 15.2 summarizes the main application areas of term rewriting, Section 15.3 gives an annotated list of selected languages and systems, and Section 15.4 surveys implementation techniques.

We have freely used the information from the various web pages. Dershowitz and Vigneron [2001] and Visser et al. [2001] have been especially useful.

15.2. Application Areas

The main application areas of term rewriting and selected systems in each area are listed in Table 15.1. Note that a system may have several application areas. Systems in bold face are discussed in more detail in Section 15.3. The other ones are only mentioned in this section. We briefly describe each area.

Computer algebra deals with symbolic computation in mathematics. Rewriting is used for the transformation and simplification of mathematical expressions. An introduction to the algorithmic basis of the mathematical engine in computer algebra systems is given in von zur Gathen and Gerhard [1999].

Functional languages have their basis in the lambda calculus (Chapter 10). Unlike imperative programming languages, they do not allow multiple assignment and are thus *side-effect free.* Clean and Haskell use lazy evaluation (Chapter 12) and graph rewriting (Chapter 13).

Algebraic specification and rewrite languages may be used to define abstract datatypes, concurrent systems, or the semantics of languages in terms of (conditional) equations/rewrite rules (Ehrig and Mahr [1985], Bergstra et al. [1989]) (see also below). Algebraic specification systems are used to prototype specifications by means of (conditional) term rewriting (Chapter 3) and to prove certain of their properties. Elan and Maude feature user-defined rewrite strategies (Chapter 9) and AC-rewriting (Chapter 14).

Specification of concurrent systems is a first step towards their analysis and verification. The concurrency aspects may be specified using a form of process algebra (Bergstra et al. [2001]). In that case, term rewriting is used to normalize the corresponding process expressions.

Language definition aims at specifying the syntax and semantics of programming and application languages. Using algebraic semantics (Chapter 7, Bergstra et al. [1989], Goguen and Malcolm [1996]), the typechecking, evaluation, translation, transformation, and other semantic aspects of a language are defined using equations or rewrite rules. An overview of various tool-oriented approaches to programming languages semantics is Heering and Klint [2000].

Program transformation is used for program or code optimization in preprocessors and compilers as well as for systematic program modification for purposes of software maintenance and renovation. Usually, the former is based on a cost-driven strategy (see also code generation below), while the latter is a more traditional normalization with respect to rewrite rules for style improvement or translation to another language version. There are many older program transformation systems, all of them including some form of rewriting. An overview of transformation systems is given in Visser [2001]. The HOPS program transformation system supports second-order term graph rewriting, while Stratego supports user-defined rewrite strategies (Chapter 9).

Code generation is done by compiler backends to generate instructions for a given machine. There are various code generator generators that use pattern matching and dynamic programming to compute, using a cost-driven strategy, the optimal covering of a tree with a set of patterns. Typically, pattern cost corresponds to the cost of the machine instructions generated for each pattern. The goal is to generate the cheapest code. All approaches use a table generation phase for efficient pattern matching. They differ in the way dynamic programming is handled, either during generation of the code generator (BURG) or during use of the code generator (TWIG, BEG).

Theorem proving uses term rewriting and narrowing (Chapter 7) to handle equality and to solve equations. In this context, term rewriting is often called *demodulation*, while narrowing is called *paramodulation*. A *demodulator* is a rewrite rule. This brings in the issue of completion (Chapter 7) and termination proofs (Chapter 6), and some of these systems (e.g., Otter) offer facilities for this. Others just include a basic rewriting capability. The generic theorem proving environment Isabelle supports higher-order rewriting in the sense of Pattern Rewrite Systems (Nipkow and Prehofer [1998]) or HRSs as they are called in Chapter 11. Inductive proof by consistency (Chapter 7) is, for instance, supported by RRL.

Rewriting workbenches provide tools to create, check, analyze, and complete (Chapter 7) term rewriting systems. In addition, they may contain facilities for automatic or interactive proof and proof checking.

Application area	**Selected systems**
Computer algebra	Maple (Waterloo Maple, Inc. [2001]), Mathematica (Wolfram Research, Inc. [2001]), MatLab (MathWorks, Inc. [2001]), Reduce (Cologne University [2001])
Functional languages	**Clean**, **Haskell**, SML (Bell Laboratories [2001])
Algebraic specification and rewrite languages	**ASF+SDF**, **CafeOBJ**, **Elan**, **LP**, **Maude**, **OBJ3**
Specification of concurrent systems	**LOTOS**, **Maude**, **μCRL**, **PSF**
Language definition	**ASF+SDF**, **OBJ3**, Software Refinery (Markosian et al. [1994])
Program transformation	**ASF+SDF**, **HOPS**, Software Refinery (Markosian et al. [1994]), **Stratego**, TAMPR (Boyle [1989], Boyle et al. [1997]), **TXL**
Code generation	**BEG**, Burg (Fraser, Hanson and Proebsting [1992], Fraser, Henry and Proebsting [1992], Proebsting [1995]), TWIG (Aho et al. [1989])
Theorem proving	**ACL2**, AUTOMATH (de Bruijn [1980]), **CafeOBJ**, **Coq**, **daTac**, **EQP**, **Elan**, **Isabelle**, **LP**, **Maude**, **Nqthm**, **OBJ3**, **OSHL**, **Otter**, **RRL**, **PVS**, **SPIKE**
Rewriting workbenches	**HOPS**, **ReDuX**, Reve (Forgaard and Guttag [1984]), **RRL**

Table 15.1: Main application areas of rewriting and selected systems

15.3. Systems

Name: ACL2 (A Computational Logic for an Applicative Core Language).
Features: Logic based on an applicative subset of Common Lisp, explicit rewrite operation, conditional rewriting.
Applications: Theorem proving, symbolic execution, model checking, proof checking.
Web site: UT Austin [2001a].
Notes: ACL2 is both a programming language in which one can model computer systems and a tool to help prove properties of those models. It is based on and extends the Boyer-Moore theorem prover Nqthm (see below). The ACL2 approach itself is presented in Kaufmann et al. [2000b] and case studies are described in Kaufmann et al. [2000a].

Name: ASF+SDF (Algebraic Specification Formalism + Syntax Definition Formalism).
Features: User-definable lexical and context-free syntax, general context-free parsing, rewriting with both positive and negative conditions, list matching, traversal functions, interpreter and compiler, parameterized modules.
Applications: Language prototyping, tool generation, domain-specific language development, program transformation, software renovation.
Web site: CWI [2001].
Notes: The ASF+SDF Meta-Environment is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language. The generation process is controlled by a specification of the target language expressed in the ASF+SDF metalanguage, which typically defines such features as syntax, prettyprinting, typechecking, and execution or transformation of programs in the target language (van den Brand et al. [2001]).

Name: BEG (Backend Generator).
Features: Tree pattern matching, dynamic programming.
Applications: Code generation.
Web site: FIRST [2001].
Notes: BEG generates compiler backends from a declarative specification of the code generation process. The generated backends use tree pattern matching for code selection. Machine instructions are described by tree patterns and code generation is equivalent to finding a cover of the input tree using these tree patterns. Cost values associated with the instruction patterns are used to indicate desired qualities of the code, e.g., speed or code size. Ambiguities during selection of a cover are resolved in favor of the cheapest cover. BEG has been used to generate various code generators including a just-in-time code generator for Java. Related systems using similar techniques are BURG (Fraser, Hanson and Proebsting [1992], Fraser, Henry and Proebsting [1992], Proebsting [1995]) and TWIG (Aho et al. [1989]).

Name: CafeOBJ.
Features: Rewriting logic, hidden algebras, object-oriented modeling, distributed specification development environment, proof assistant.
Applications: Software specification.
Web site: Language Design Lab [2001].
Notes: Cafe is the name of an environment for the systematic development of formal specifications based on algebraic specification techniques. It is designed to support semantic representations of problems and reasoning methods not provided by current CASE tools. CafeOBJ is a successor to OBJ3 (see below). One of the most important features of CafeOBJ is its

support for object-oriented modeling (OOM). A larger scale project for developing a network-based unified environment for CafeOBJ including a verifier/checker, browser/editor, and library/cases is in progress.

Name: Clean.
Features: Lazy higher-order functional language, term graph rewriting, pattern matching, polymorphic typing, uniqueness types.
Applications: General programming.
Web site: Nijmegen University [2001].
Notes: Clean and Haskell (see below) are both based on (term) graph rewriting. In the case of Clean, the semantics of the language itself is based on term graph rewriting and precisely describes the representation and sharing of nodes. As a result, destructive updates of unshared nodes are possible without destroying the functional behaviour of the language. In the case of Haskell, different implementations may treat the sharing of nodes differently, and graph rewriting is only a matter of efficiency.

Name: Coq.
Features: Calculus of (co)inductive constructions, Gallina specification language.
Applications: Proof assistant.
Web site: INRIA Rocquencourt [2001].
Notes: Coq allows mechanical proof checking of assertions in the construction calculus, interactive construction of formal proofs, and extraction of a program from the constructive proof of its formal specification.

Name: daTac.
Features: Associative/commutative deduction with constraints, paramodulation.
Applications: Verification of cryptographic protocols.
Web site: LORIA Nancy [2001a].
Notes: The aim of the daTac system is to do automated deduction in first-order logic with equality. Its specialty is to apply deductions modulo some equational properties of operators, such as commutativity (C) or associativity-commutativity (AC). The implemented deduction techniques, based on resolution, paramodulation and term rewriting, are refutationally complete.

Name: Elan.
Features: Rewriting logic language, definable rewriting strategies, AC-rewriting, interpreter and compiler.
Applications: Theorem proving, constraint solving.
Web site: LORIA Nancy [2001b].
Notes: The Elan system (Borovanský et al. [2001]) provides an environment

for specifying and prototyping deduction systems in a language based on rewrite rules controlled by user-defined strategies. It is based on the rewriting calculus and on rewriting logic and offers a simple logical framework for the combination of computation and deduction paradigms. Elan can support the design of theorem provers, logic programming languages, constraint solvers, and decision procedures. Furthermore, it offers a modular framework for studying their combination.

Name: EQP (Equational Prover).
Features: AC-unification and -matching, paramodulation.
Applications: Theorem Proving for lattice-like structures, Robbins conjecture.
Web site: Argonne National Laboratory [2001a].
Notes: EQP is an automated theorem proving program for first-order equational logic. Its strengths are good implementations of associative-commutative unification and matching, a variety of strategies for equational reasoning, and fast search. It is well-suited for problems about lattice-like structures. EQP originates from the same group that has developed Otter (see below).

Name: Haskell.
Features: Lazy higher-order functional language, graph rewriting, pattern matching, polymorphic typing, monads, modules.
Applications: General programming.
Web site: Yale University [2001].
Notes: Haskell has been used in applications ranging from an equational reasoning assistant to speech recognition. One method of implementing or prototyping domain-specific languages is to embed them in Haskell. Instead of implementing an interpreter or compiler in it, Haskell is enriched by libraries for domain specific datatypes and functions that turn Haskell into a domain specific language. For further comments, see the above discussion on graph rewriting for Clean.

Name: HOPS (Higher Object Programming System).
Features: Term graphs, second-order term graph rewriting, editing and visualization of term graphs.
Applications: Program Transformation.
Web site: Universität der Bundeswehr München [2001].
Notes: HOPS is a graphical, interactive, program development and program transformation system based on acyclic term graphs. Unlike most systems that take a textual representation as starting point, in HOPS programs are made up of term graphs. In term graphs, the structure is exposed in an explicit manner, and interaction is guided by this structure.

Name: Isabelle.
Features: Generic theorem prover/logic framework, higher-order meta-logic, higher-order term rewriting in the sense of Pattern Rewrite Systems or HRSs, type classes.
Applications: Development of logics.
Web site: Cambridge University [2001].
Notes: Isabelle has been instantiated to support reasoning in various logics: both classical and constructive first-order logic, higher-order logic, ZF, a version of type theory, and various modal logics, among others.

Name: LP (Larch Prover).
Features: Many-sorted first-order logic, unconditional rewriting.
Applications: Interactive theorem prover, reasoning about circuit design, concurrent algorithms, hardware, and software.
Web site: MIT [2001].
Notes: The philosophy of LP is based on the observation that initial attempts to state interesting conjectures correctly, and then prove them, hardly ever succeed on the first try. As a result, LP is designed to assist in reasoning by carrying out routine (and possibly lengthy) steps in a proof automatically and by providing useful information about why proofs fail, if and when they do. Because conjectures are often flawed, LP is not designed to find difficult proofs automatically. For example, LP does not use heuristics to formulate additional conjectures that might be useful in a proof. Instead, LP makes it easy for users to employ standard techniques such as simplification and proofs by cases, induction, and contradiction, either to construct proofs or to understand why they have failed. LP also provides support for rechecking proofs following changes in axioms, conjectures, or proof strategies. This support also promotes proof re-use: users can edit old proofs to prove new conjectures, and LP will check that the progress of the proof stays “in sync” with the users’ intentions.

Name: LOTOS.
Features: Process expressions, abstract datatypes.
Applications: Communication protocols.
Web site: Twente University [2001].
Notes: LOTOS is a specification language for describing the behaviour of concurrent systems. It consists of a sublanguage for process definition that contains features from CSP (Hoare [1985]) and CCS (Milner [1980]) and a sublanguage for defining abstract datatypes based on ACT-ONE (Ehrig and Mahr [1985]). LOTOS has played a prominent role in the standardization of communication protocols but has also been used for modeling various phases in the software engineering life cycle. Several tools exist for simulating the behaviour of the specified systems. They incorporate limited term rewriting functionality.

Name: Maude.
Features: Rewriting logic language, reflection, AC-rewriting.
Applications: Logical framework, models of computation, theorem provers.
Web site: SRI [2001a].
Notes: Maude is a reflective language and system supporting both equational and rewriting logic specification and programming. Maude has been influenced by the OBJ3 language (see below), which can be regarded as an equational logic sublanguage. Besides supporting equational specification and programming, Maude also supports rewriting logic computation. Rewriting logic is a logic of concurrent change that can naturally deal with state and with concurrent computations. Maude supports logical reflection. This makes Maude extensible, supports an extensible algebra of module composition operations, and allows many meta-programming and metalanguage applications.

Name: μCRL (Micro Common Representation Language).
Features: Process expressions, abstract datatypes.
Applications: Verification of concurrent systems.
Web site: University of Amsterdam [2001a].
Notes: The Common Representation Language (CRL) is intended as a universal intermediate language for specification languages: it contains features of current specification languages and each of these languages could be translated to CRL. In this way, CRL-based tools become available for all these languages. μCRL has been introduced as an intermediate stage that reduces semantic complexities. It contains some essential elements of CRL, but has far less language features. It focuses on the (non-modular) specification of data and processes and enables a thorough analysis of their behaviour. This has resulted in a proof theory, based on natural deduction, that can be used to prove equality of processes and data terms in a CRL specification.

Name: Nqthm.
Features: Quantifier-free, first-order logic resembling Pure Lisp.
Applications: Proving theorems in mathematics, metamathematics, the theory of computation, and computer science.
Web site: UT Austin [2001b].
Notes: Nqthm is essentially an implementation of the “Boyer-Moore theorem prover”. It is based on a Lisp-like language and the behaviour of the prover is determined by a rule base. These rules are derived from the axioms, definitions, and theorems submitted by the user. The user can guide the prover by providing an appropriate sequence of lemmas. Applications include communication protocols, concurrent algorithms, operating systems, and hardware verification. ACL2 (see above) is a successor of Nqthm.

Name: OBJ3.
Features: Order-sorted algebra, initial algebra semantics, user-definable mixfix syntax, rewriting modulo AC, user-definable strategies, modular specifications, parameterized programming.
Applications: Software design, prototyping, theorem proving, hardware verification.
Web site: UCSD [2001].
Notes: OBJ3 is a broad spectrum algebraic programming and specification language. It is based on order-sorted algebra, which provides a formal basis for user-definable subtypes, exception handling, multiple inheritance, overloading, multiple representations, coercions, and more. OBJ3 also supports user-definable mixfix syntax, user-definable execution strategies, rewriting modulo standard equational theories (A, C, I, etc.), and memoization. OBJ3 also provides parameterized programming, with parameterized modules, module instantiation, views, module expressions, etc., to support flexible program structuring and reuse. OBJ3 is a member of the "OBJ" language family that includes OBJ2, CafeOBJ (see above), and others.

Name: OSHL (Ordered Semantic Hyper Linking).
Features: Instance-based refutation, completion with AC constraints, narrowing.
Applications: Theorem Proving.
Web site: UNC [2001].
Notes: The OSHL theorem prover is based on a first-order theorem proving strategy — ordered semantic hyper linking. This is an instance-based refutational theorem proving strategy. It solves first-order problems by reducing them to propositional problems, and it uses an efficient propositional decision procedure. It uses natural semantics of an input problem to guide its search. It also incorporates term rewriting to handle equality. The propositional efficiency, semantic guidance and equality support allow OSHL to solve problems that are difficult for many other strategies.

Name: Otter.
Features: First-order logic with equality, demodulation/paramodulation, term orderings, completion.
Applications: Problems in abstract algebra and formal logic.
Web site: Argonne National Laboratory [2001b].
Notes: Otter has been designed to prove theorems stated in first-order logic with equality. Otter's inference rules are based on resolution and paramodulation, and it includes facilities for term rewriting, term orderings, Knuth-Bendix completion, weighting, and strategies for directing and restricting searches for proofs. Otter can also be used as a symbolic calculator and has an embedded equational programming system. Currently, the

main application of Otter is research in abstract algebra and formal logic. Otter and its predecessors have been used to answer many open questions in the areas of finite semigroups, ternary Boolean algebra, logic calculi, combinatory logic, group theory, lattice theory, and algebraic geometry.

Name: PSF (Process Specification Formalism).
Features: Process expressions, abstract datatypes.
Applications: Specification and simulation of concurrent systems.
Web site: University of Amsterdam [2001b].
Notes: PSF is a formal description technique developed for the specification of concurrent systems. PSF has been designed as the basis for a set of tools to support the process algebra formalism ACP (Bergstra et al. [2001]). It is very close to the informal syntax normally used in denoting ACP-expressions. The part of PSF that deals with the description of data is based on ASF (Bergstra et al. [1989]). PSF supports the modular construction of specifications and the parameterization of modules.

Name: PVS (Prototype Verification System).
Features: Typed higher-order logic, propositional and quantifier rules, induction, rewriting, decision procedures for linear arithmetic.
Applications: Formalization of requirements and design-level specifications, analysis of intricate and difficult problems.
Web site: SRI [2001b].
Notes: PVS and its predecessors have been chiefly applied to algorithms and architectures for fault-tolerant flight control systems, and to problems in hardware and real-time system design. It is optimized for large proofs.

Name: ReDuX.
Features: Many-sorted, first-order, algebraic specifications, completion, term orderings.
Applications: Workbench for term rewriting, mathematical problems, hardware verification.
Web site: University of Tübingen [2001].
Notes: ReDuX is a work-bench for programming and experimenting with term rewriting systems. It is oriented towards the implementation of completion procedures. In particular, it provides Knuth-Bendix completion, Peterson-Stickel completion modulo associativity and commutativity, and inductive completion. Another important feature of ReDuX is that it supports fast rewriting for specifications containing standard datatypes.

Name: RRL (Rewrite Rule Laboratory).
Features: First-order logic with equality, completion, proof by consistency, cover sets.
Applications: Software verification, protocol verification, mathematical

problems.
Web site: University of Iowa [2001].
Notes: RRL is an automated reasoning system based on rewriting techniques. It has implementations of completion procedures for generating a complete set of rewrite rules from an equational axiomatization, associative-commutative matching and unification, algorithms for orienting equations into terminating rewrite rules, refutational methods for first-order theorem proving and methods for proving first-order equational formulas by induction. RRL has been used to solve mathematical problems in automated reasoning and has been applied to investigate the use of formal methods in hardware and software design.

Name: SPIKE.
Features: Induction based on AC matching, lexicographic path ordering, simultaneous induction, saturation, paramodulation, superposition, sufficient completeness.
Applications: Circuit verification, program verification.
Web site: LORIA Nancy [2001c].
Notes: SPIKE performs proofs in theories whose axioms are first-order conditional equations. The equations are oriented as rewrite rules using the lexicographic path ordering. Systems of rewrite rules are completed by means of saturation, an extension of Knuth-Bendix completion for conditional rules. The primary focus is on the fully automatic proof or refutation of inductive theorems in conditional theories.

Name: Stratego.
Features: Single-sorted rewrite rules, programmable rewriting strategies, interpreter and compiler.
Applications: Program transformation, compiler construction.
Web site: Utrecht University [2001].
Notes: In Stratego, basic transformation rules are expressed by means of labeled rewrite rules. Exhaustively applying all rewrite rules in a collection of valid rules is often not desirable. A system of rules can be non-terminating, or, more frequently, non-confluent. To control the application of transformation rules, Stratego provides a language for defining rewriting strategies based on primitives for sequential programming and abstract syntax tree traversal. A rewriting strategy selects a number of rules from the available rules and defines in what order these rules are applied to a program fragment.

Name: TXL (Transformation Language).
Features: Context-free grammars, transformation rules based on pattern/replacement pairs, programmable application of transformation rules.
Applications: Programming language processing, software engineering and renovation, database query optimization and schema translation.

Web site: The TXL Company [2001].
Notes: The TXL programming language is a hybrid functional/rule-based language with unification, implied iteration, and deep pattern matching. Each TXL program has two components: (1) a description of the structures to be transformed specified as an EBNF grammar, and (2) a set of structural transformation rules specified by example, using pattern/replacement pairs. TXL has similarities with rewriting but is actually a pattern-based tree transformation language that allows full programming of tree traversals and operations.

15.4. Implementation of Term Rewriting

The following brief account is more or less chronological, but does not make claims to completeness. An earlier survey of TRS implementation techniques is Bouma and Walters [1989]. A standard reference on (lazy) functional language implementation is Peyton Jones [1987]. Many of the functional and rewrite language implementations in existence at the time of its publication are compared in Hartel et al. [1996].

A top-down matching algorithm that reduces first-order term matching to string matching is given in Hoffmann and O'Donnell [1982]. Basically, the set of left-hand sides of a TRS is transformed into a deterministic finite automaton (DFA). This approach was used in the implementation of the Equation Interpreter (O'Donnell [1985]), an equational programming system featuring parallel outermost reduction, a normalizing strategy for orthogonal rewriting systems (Chapter 4). DFA code for term matching is generated by most current rewrite language compilers. Optimization of the matching automaton under a left-to-right constraint is discussed in Nedjah et al. [1997].

Translation of first-order term rewriting systems to Prolog has been used to prototype algebraic specification formalisms in terms of rewriting. Various translation schemes are compared in Drosten [1988]. Term rewriting is obtained by term decomposition and unification, but the full power of unification is not needed and the resulting Prolog code is inefficient.

In the same vein, translation of higher-order term rewriting systems (Chapter 11) to λProlog is discussed in Felty [1992], Heering [1992]. λProlog is an extension of Prolog to typed λ-terms (Nadathur and Miller [1988]). Basically, the functions declared in a λProlog program generate a domain of polymorphically typed λ-terms, and polymorphic higher-order unification takes the place of first-order unification in the proof procedure.

Working directly with Lisp function calls rather than Prolog unification, compilation of first-order (conditional) term rewriting systems to Lisp (Kaplan [1987]) in the Asspegique algebraic specification environment (Bidoit and Choppy [1985]) was a step forward. Oriented towards an innermost reduction strategy, each function in the signature of the TRS is compiled to a Lisp function whose body contains the matching conditions for the rewrite rules defining the function in the TRS. The generated Lisp code is then fed to a Lisp compiler for further code generation. With various refinements and different target languages (in particular C), this basic scheme is still used in current compilers.

Some compilers generate abstract machine code. The lazy functional language Clean, for instance, is compiled to code for the ABC abstract graph rewriting machine (Plasmeijer and van Eekelen [1993]). Graph rewriting (Chapter 13) is widely used as a method for improving the space and time efficiency of functional language implementations.

The equational programming language Epic is compiled to code for the Abstract Rewrite Machine (ARM) (Fokkink et al. [1998]). Based on the notion of *minimal term rewriting systems* (MTRS), compilation to ARM code basically amounts to bringing rewrite rules in so-called *minimal* form. A compiler for the lazy functional language Haskell generating extended Microsoft Intermediate Language (MS-ILX) is described in Syme [2001].

Because of its flexibility and wide availability, and despite some shortcomings (Peyton Jones et al. [1998]), C has become a *de facto* standard target for code generation. Folk wisdom has it that C code is 2–3 times slower than native code, but this is not borne out by the "Pseudoknot" benchmark results reported in Hartel et al. [1996], where the best functional language and rewrite language compilers generate C code. The probable reason is that many C compilers perform sophisticated optimizations.

Among others, the rewriting-based language definition formalism ASF+SDF (van den Brand et al. [2000]), the rewrite language Elan (Kirchner and Moreau [2001]), and the lazy functional language Haskell (Peyton Jones et al. [1993]) are compiled to C. While the Haskell compiler generates graph rewriting code, the ASF+SDF and Elan compilers use a *maximal subterm sharing* scheme that is transparent to the rewriting process (van den Brand et al. [1999]). In this scheme, known as *hash-consing* in Lisp (Allen [1978]), (sub)terms are created during rewriting only when they do not yet exist. The unique storage of terms reduces overall memory requirements and allows structural equality checks to be replaced by pointer equality checks, thus improving execution speed.

Some rewrite languages, among them Elan and Maude, support associative-commutative rewriting. ASF+SDF features list rewriting, a special case of associative (or string) rewriting. Both are examples of rewriting modulo an equivalence as discussed in Chapter 14, Section 14.3. While term matching is linear, both AC- and A-matching are NP-complete (Benanav et al. [1985]). A general AC-matching algorithm is given in Eker [1995], but since the actual patterns to be matched are known at compile time, the Elan compiler is often able to generate specialized AC-matching code that is more efficient. Similarly, the ASF+SDF compiler can often generate specialized list matching code. Nevertheless, AC and list rewriting remain expensive features that should be used with restraint.

Bibliography

Aho, A.V., M. Ganapathi and S.W.K. Tjiang [1989]. Code generation using tree matching and dynamic programming, *ACM Transactions on Programming Languages and Systems* **11**, pp. 491–516.

Allen, J.R. [1978]. *Anatomy of Lisp*, McGraw-Hill.

Argonne National Laboratory [2001a]. EQP Equational Prover. `http://www-unix.mcs.anl.gov/AR/eqp/`.

Argonne National Laboratory [2001b]. Otter: An automated deduction system. `http://www-unix.mcs.anl.gov/AR/otter/`.

Bell Laboratories [2001]. Standard ML of New Jersey. `http://cm.bell-labs.com/cm/cs/what/smlnj/`.

Benanav, D., D. Kapur and P. Narendran [1985]. Complexity of matching problems, *in:* J.-P. Jouannaud (ed.), *Rewriting Techniques and Applications (RTA '85)*, Lecture Notes in Computer Science 202, Springer-Verlag, pp. 417–429.

Bergstra, J.A., J. Heering and P. Klint (eds.) [1989]. *Algebraic Specification*, ACM Press/Addison-Wesley.

Bergstra, J.A., A. Ponse and S.A. Smolka [2001]. *Handbook of Process Algebra*, Elsevier.

Bidoit, M. and C. Choppy [1985]. Asspegique: An integrated environment for algebraic specifications, *in:* H. Ehrig et al. (eds.), *Formal Methods and Software Development (TAPSOFT '86)*, Lecture Notes in Computer Science 186, Springer-Verlag, pp. 246–260.

Borovanský, P., H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen and M. Vittek [2001]. *ELAN User Manual*, v3.4 edition, LORIA Nancy, Nancy, France.

Bouma, L.G. and H.R. Walters [1989]. Implementing algebraic specifications, *in:* J.A. Bergstra, J. Heering and P. Klint (eds.), *Algebraic Specification*, ACM Press/Addison-Wesley, pp. 199–282.

Boyle, J.M. [1989]. Abstract programming and program transformation — An approach to reusing programs, *in:* T.J. Biggerstaff and A.J. Perlis (eds.), *Software Reusability*, Vol. 1, ACM Press, pp. 361–413.

Boyle, J.M., T.J. Harmer and V.L. Winter [1997]. The TAMPR program transformation system: Simplifying the development of numerical software, *in:* E. Arge, A.M. Bruaset and H.P. Langtangen (eds.), *Modern Software Tools in Scientific Computing (SciTools '96)*, Birkhauser, pp. 353–372.

van den Brand, M.G.J., J. Heering, P. Klint and P.A. Olivier [2000]. Compiling language definitions: The ASF+SDF compiler, *Technical Report SEN-R0014*, CWI. `http://www.cwi.nl/CWIreports/SEN/SEN-R0014.ps.Z`.

van den Brand, M.G.J., P. Klint and P. A. Olivier [1999]. Compilation and memory management for ASF+SDF, *in:* S. Jähnichen (ed.), *Compiler Construction (CC '99)*, Lecture Notes in Computer Science 1575, Springer-Verlag, pp. 198–213.

van den Brand, M.G.J., A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser and J. Visser [2001]. The ASF+SDF Meta-Environment: A component-based language development environment, *in:* R. Wilhelm (ed.), *Compiler Construction (CC 2001)*, Lecture Notes in Computer Science 2027, Springer-Verlag, pp. 365–370.

de Bruijn, N.G. [1980]. A survey of the project AUTOMATH, *in:* J.P. Seldin and J.R. Hindley (eds.), *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, pp. 579–606.

Cambridge University [2001]. Isabelle. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

Cologne University [2001]. Reduce. `http://www.uni-koeln.de/REDUCE/`.

CWI [2001]. ASF+SDF Meta-Environment. `http://www.cwi.nl/projects/MetaEnv/`.

Dershowitz, N. and L. Vigneron [2001]. Rewriting Home Page. `http://www.loria.fr/~vigneron/RewritingHP/`.

Drosten, K. [1988]. Translating algebraic specifications to Prolog programs: A comparative study, *in:* J. Grabowski, P. Lescanne and W. Wechler (eds.), *Algebraic and Logic Programming (ALP '88)*, Lecture Notes in Computer Science 343, Springer-Verlag, pp. 137–146.

Ehrig, H. and B. Mahr [1985]. *Fundamentals of algebraic specifications I*, EATCS Monographs on Theoretical Computer Science 6, Springer-Verlag.

Eker, S.M. [1995]. Associative-commutative matching via bipartite graph matching, *Computer Journal* **38**, pp. 381–399.

Felty, A. [1992]. A logic programming approach to implementing higher-order term rewriting, *in:* L.-H. Eriksson, L. Hallnäs and P. Schroeder-Heister (eds.), *Extensions of Logic Programming (ELP '91)*, Lecture Notes in Artificial Inteligence 596, Springer-Verlag, pp. 135–158.

FIRST [2001]. The backend generator BEG. `http://www.first.gmd.de/beg/`.

Fokkink, W.J., J.F.Th. Kamperman and H.R. Walters [1998]. Within ARM's reach: Compilation of left-linear rewrite systems via minimal rewrite systems, *ACM Transactions on Programming Languages and Systems* **20**, pp. 679–706.

Forgaard, R. and J.V. Guttag [1984]. REVE: A term rewriting system generator with failure-resistant Knuth-Bendix, *in:* J.V. Guttag, D. Kapur and D. Musser (eds.), *Proceedings of the NSF Workshop on the Rewrite Rule Laboratory*, General Electric Corporate Research and Development Center, Technical Report 84GEN008, pp. 5–32.

Fraser, C.W., D.R. Hanson and T.A. Proebsting [1992]. Engineering a simple, efficient code-generator generator, *ACM Letters on Programming Languages and Systems* **1**, pp. 213–226.

Fraser, C.W., R.R. Henry and T.A. Proebsting [1992]. BURG — Fast optimal instruction selection and tree parsing, *ACM SIGPLAN Notices* **27**(4), pp. 68–76.

von zur Gathen, J. and J. Gerhard [1999]. *Modern Computer Algebra*, Cambridge University Press.

Goguen, J.A. and G. Malcolm [1996]. *Algebraic Semantics of Imperative Programs*, MIT Press.

Hartel, P.H. et al. [1996]. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark, *Journal of Functional Programming* **6**, pp. 621–655.

Heering, J. [1992]. Implementing higher-order algebraic specifications, *in:* D. Miller (ed.), *Proceedings of the Workshop on the λProlog Programming Language*, University of Pennsylvania, Philadelphia, Technical Report MS-CIS-92-86, pp. 141–157. `http://www.cwi.nl/~jan/HO.WLP.ps`.

Heering, J. and P. Klint [2000]. Semantics of programming languages: A tool-oriented approach, *ACM SIGPLAN Notices* **35**(3), pp. 39–48.

Hoare, C.A.R. [1985]. *Communicating Sequential Processes*, Prentice-Hall International.

Hoffmann, C.M. and M.J. O'Donnell [1982]. Pattern matching in trees, *Journal of the ACM* **29**, pp. 68–95.

INRIA Rocquencourt [2001]. The Coq proof assistant. `http://pauillac.inria.fr/coq/`.

Kaplan, S. [1987]. A compiler for conditional term rewriting systems, *in:* P. Lescanne (ed.), *Rewriting Techniques and Applications (RTA '87)*, Lecture Notes in Computer Science 256, Springer-Verlag, pp. 25–41.

Kaufmann, M., P. Manolis and J. Strother Moore [2000a]. *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Publishers.

Kaufmann, M., P. Manolis and J. Strother Moore [2000b]. *Computer-Aided Reasoning: An Approach*, Kluwer Academic Publishers.

Kirchner, H. and P.-E. Moreau [2001]. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories, *Journal of Functional Programming* **11**, pp. 207–251.

Language Design Lab [2001]. CafeOBJ. `http://www.ldl.jaist.ac.jp/cafeobj/`.

LORIA Nancy [2001a]. daTac. `http://www.loria.fr/equipes/protheo/SOFTWARES/DATAC/`.

LORIA Nancy [2001b]. Elan. `http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/`.

LORIA Nancy [2001c]. SPIKE. `http://www.loria.fr/equipes/protheo/SOFTWARES/SPIKE/`.

Markosian, L., P. Newcomb, R. Brand, S. Burson and T. Kitzmiller [1994]. Using an enabling technology to reengineer legacy systems, *Communications of the ACM* **37**, pp. 58–70.

MathWorks, Inc. [2001]. MatLab. `http://www.mathworks.com/products/matlab/`.

Milner, R. [1980]. *A Calculus of Communicating Systems*, Springer-Verlag.

MIT [2001]. LP. `http://www.sds.lcs.mit.edu/Larch/LP/overview.html`.

Nadathur, G. and D. Miller [1988]. An overview of λProlog, *in:* R.A. Kowalsi and K.A. Bowen (eds.), *Logic Programming — Proceedings of the Fifth International Conference and Symposium*, Vol. 1, MIT Press, pp. 810–827.

Nedjah, N., C.D. Walter and S.E. Eldridge [1997]. Optimal left-to-right pattern-matching automata, *in:* M. Hanus, J. Heering and K. Meinke (eds.), *Algebraic and Logic Programming (ALP '97/HOA '97)*, Lecture Notes in Computer Science 1298, Springer-Verlag, pp. 273–286.

Nijmegen University [2001]. Clean. `http://www.cs.kun.nl/~clean/`.

Nipkow, T. and C. Prehofer [1998]. Higher-order rewriting and equational reasoning, *in:* W. Bibel and P. Schmitt (eds.), *Automated Deduction — A Basis for Applications. Volume I: Foundations*, Applied Logic Series 8, Kluwer, pp. 399–430.

O'Donnell, M.J. [1985]. *Equational Logic as a Programming Language*, MIT Press.

Peyton Jones, S.L. [1987]. *The Implementation of Functional Programming Languages*, Prentice-Hall International.

Peyton Jones, S.L., C.V. Hall, K. Hammond, W.D. Partain and P.L. Wadler [1993]. The Glasgow Haskell compiler: A technical overview., *Proceedings of Joint Framework for Information Technology Technical Conference (JFIT), Keele*, DTI/SERC, pp. 249–257.

Peyton Jones, S.L., T. Nordin and D. Oliva [1998]. `C--`: A portable assembly language, *in:* C. Clack, K. Hammond and T. Davie (eds.), *Implementation of Functional Languages (IFL '97)*, Lecture Notes in Computer Science 1467, Springer-Verlag, pp. 1–19.

Plasmeijer, R. and M. van Eekelen [1993]. *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley.

Proebsting, T.A. [1995]. BURS: Automata generation, *ACM Transactions on Programming Languages and Systems* **17**, pp. 461–486.

SRI [2001a]. The Maude system. `http://maude.csl.sri.com/`.

SRI [2001b]. The PVS specification and verification system. `http://pvs.csl.sri.com/`.

Syme, D. [2001]. ILX: Extending the .NET Common IL for functional language interoperability, *Proceedings Babel '01*. `http://research.microsoft.com/~dsyme/papers/babel01.pdf`.

The TXL Company [2001]. The TXL transformation system. `http://www.txl.ca/`.

Twente University [2001]. LOTOS. `http://wwwtios.cs.utwente.nl/lotos/`.

UCSD [2001]. The OBJ family. `http://www.cse.ucsd.edu/users/goguen/sys/obj.html`.

UNC [2001]. OSHL. `http://www.cs.unc.edu/~zhu/prover.html`.

Universität der Bundeswehr München [2001]. HOPS. `http://ist.unibw-muenchen.de/kahl/HOPS/`.

University of Amsterdam [2001a]. The μCRL programming environment project. `http://www.science.uva.nl/pub/programming-research/software/muCRL/`.

University of Amsterdam [2001b]. PSF Process Specification Formalism. `http://adam.wins.uva.nl/~psf/`.

University of Iowa [2001]. RRL. `http://www.cs.uiowa.edu/~hzhang/induc.html`.

University of Tübingen [2001]. ReDuX. `http://www-sr.informatik.uni-tuebingen.de/~buendgen/redux.html`.

UT Austin [2001a]. ACL2. `http://www.cs.utexas.edu/users/moore/acl2/`.

UT Austin [2001b]. Nqthm, the Boyer-Moore prover. `ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/index.html`.

Utrecht University [2001]. Stratego. `http://www.stratego-language.org`.

Visser, E. [2001]. A survey of strategies in program transformation systems, *in:* B. Gramlich and S. Lucas Alba (eds.), *Proceedings of the Workshop on Reduction Strategies in Rewriting and Programming (WRS-01)*, Electronic Notes in Theoretical Computer Science 57, Elsevier.

Visser, E. et al. [2001]. Program-Transformation.Org. `http://www.program-transformation.org/`.

Waterloo Maple, Inc. [2001]. Maple. `http://www.maplesoft.com/`.

Wolfram Research, Inc. [2001]. Mathematica. `http://www.wolfram.com/`.

Yale University [2001]. Haskell. `http://www.haskell.org/`.