Generalized Rewriting in Type Theory*

David A. Basin
Max-Planck-Institut für Informatik
Saarbrücken, Germany
Email:basin@mpi-sb.mpg.de

Abstract

While type theories such as Nuprl are expressive logics for theorem proving, they present difficulties for designers of term rewriting systems. The two most serious difficulties are: 1) They do not provide a global equality. Instead users rewrite over arbitrary user-defined relations. 2) Each rewrite step must be proved valid. In general, these proofs cannot be recursively generated.

We have overcome these difficulties and designed a package for the Nuprl system that works well in practice. Our solution is an extensible set of functions for directing and validating relational inferences. The heart of our package is a set of operators that use a user-supplied lemma database to create new rewrites from old ones. These routines place no restrictions on relations; a rewrite's success depends on the strength of the database. Overall, the package allows rewrites to be pieced together in numerous ways, providing the user with a tool to construct sophisticated rewrite strategies.

1 Introduction

Our research addresses rewriting in Nuprl [4], a sequent calculus formulation of a constructive type theory similar to Martin-Löf's [14]. While most systems assume that their rewrite functions are reliable, in Nuprl, every rewrite must be proven correct. This is difficult as Nuprl's expressive power yields undecidable typing problems: There is no effective procedure that determines if a term belongs to a given type, or determines the type of a given term. As a result it is undecidable whether even simple rewrites are valid. For example, each type comes with its own equality and a proof of $t =_T t'$[1] necessitates a proof that both t and t' are members of T. Furthermore, the validity of standard congruence (substitutability) reasoning must be proven. If $t =_T t'$ then to substitute t' for t in $B[t]$ one must show that for every x and y equal in T, $B[x]$ and $B[y]$ are equal types. This too is undecidable. We explore these problems and their implications for rewriting in Section 2.

Despite these difficulties, we have implemented a rewrite package that works well in practice. Our approach provides operators that construct *relational conversions*. Given a sequent p and a term t, a conversion yields a triple: a relation R, a term t', and an ML program called a *tactic*, which should prove $t\ R\ t'$ under the assumptions in p. As with tactics, a conversion may fail; but if it succeeds, then $t\ R\ t'$ may be used as an assumption.

Basic conversions may be constructed from lemmas of the appropriate form (which is roughly that of a universally quantified chain of implications that ends

*This research was funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. The responsibility for the contents lies with the author.

[1] We shall follow the following syntactic conventions: $T, A, B, \dots$ shall represent types; Q, R, and S relations; $t, r, s, \dots$ terms and $x, y, z, \dots$ variables.

in a relation $t\ R\ t'$). We provide an operator that takes a lemma and a relation and constructs a conversion which, if it succeeds, rewrites an instance of the left-hand side of the relation in the lemma into a corresponding instance of the right hand side. Conversions themselves are data-values, and higher-order combinators provide a means to compose them and form new conversions. For example, *THENC* is a combinator that sequentially composes two conversions and *SubConv* applies a conversion to the immediate subterms of a term. A user-supplied lemma database is used in the construction of tactics that prove these rewrites valid. *SubConv*, for example, uses this database to produce a tactic that generates congruence proofs. No restrictions are placed on relations; they need not even be equivalence relations. However, whether a conversion succeeds depends on the strength of the database. The overall package is highly modular; conversions can be put together in numerous ways, allowing the user to construct sophisticated rewrite strategies. Section 3 details our implementation and illustrates the power of our approach with examples.

Our package is similar to Paulson's [16][2]. Both are collections of ML programs that can be applied in a modular higher-order style and many of the combinators are functionally identical. However, our package differs in two important respects. First, Paulson's package allows rewriting only over two congruence relations: term equality and formula equivalence. These are special cases of our general relational approach. Second, he provides separate notions of conversions (*conv* and *fconv*) for each relation, each with its own fixed strategy for proving rewrites valid. Our implementation allows the user to extend relational inference simply by expanding the lemma database. Of course, such extensibility is necessary in light of undecidable typing, but we have found the resulting generality useful in practice.

Our package has been used successfully by the author and other researchers at Cornell working in hardware verification. We have constructed a variety of rewrite procedures including a predicate calculus simplifier and associative/commutative term normalizers. We have also explored rewriting in constructive set theory. In this domain, set equality is a user defined predicate. Congruence reasoning requires proofs that each set theoretic operator respects this predicate, and these proofs must be composed to justify each rewrite. Without a package such as ours, the burden of constructing such justifications by hand would be unmanageable.

Our approach to rewriting has applications outside of Nuprl and the related theories of Martin-Löf. Logics with decidable typing but weak equality, such as the Calculus of Constructions [5], also require that users define their own equalities and prove their rewrites valid. We believe that as such logics become more popular, extensible approaches to rewriting like ours will become integrated into their use.

2 Nuprl

In this section, we highlight those aspects of Nuprl relevant to rewriting. The interested reader may consult [4] for a more complete account.

The basic objects of reasoning in Nuprl are types and members of types. The rules of Nuprl deal with sequents, objects of the form

$$x_1 : H_, x_2 : H_2,\ \ldots,\ x_n : H_n \vdash G\,.$$

To judge a sequent true essentially means that when given members x_i of H_i, one can construct an inhabitant of the goal G. Nuprl's rules are applied in a top-down fashion. That is, they allow us to refine a goal, such that a proof of the goal may be constructed from proofs of the subgoals. Nuprl provides two kinds of inference rules:

[2]It also bears similarities to ideas of Howe and Steeg [10] who implemented rewriting in Nuprl over the (type parametrized) equality relation.

primitive rules and ML programs called tactics. Nuprl tactics are similar to those in LCF [7]: Given a sequent as input, they apply primitive inference rules and other tactics to the sequent. Tactics serve as derived inference rules; their correctness is justified by the way the type structure of ML is used.

Nuprl's type theory is expressive; its intent is to facilitate the formalization of constructive mathematics. Types are stratified into a cumulative hierarchy of universes. A term is a well-formed type if and only if it inhabits some universe U_i. Type constructors include dependent function space, dependent product, disjoint union, equality, set type, and quotient type. Each type comes with its own equality relation. The equality type is a three place relation $t_1 =_T t_2$ that is inhabited exactly when t_1 and t_2 are equal members of T. Propositions are represented via the propositions-as-types correspondence: A proposition is true if and only if the type associated with it is inhabited.

A special case of type checking in Nuprl is deciding if a program meets its specification. As a result, a number of typing problems are undecidable. These include: type membership ($t \in T$); type inference; and type well-formedness ($T \in U_i$). The first two problems imply that there is no uniform way of proving the identity rewrite valid. Given a term t, we cannot infer a type T it inhabits; even if we are given a T, we cannot prove $t =_T t$ as that requires a proof of $t \in T$. Type well-formedness is a semantic property as there are syntactically legitimate yet meaningless terms. As we shall see in the next section, proofs of certain well-formedness goals may be thought of as proofs of relational congruence.

3 Generalized Rewriting

Rewriting is the process of finding a term t' that is somehow simpler than a given term t. Moreover, the terms must stand in some relation. Our rewrite functions are called conversions. Given a sequent p and a term t, a conversion returns a *rewrite triple*: a relation R, a term t', and a tactic tac that proves the assertion $t\ R\ t'$ under the assumptions in p. We call R the *rewrite relation*. We take an abstract view of relations. A relation is specified by a constructor that maps sequents and pairs of terms to terms, and a destructor that breaks down terms into pairs. Aside from the ability to construct and destruct relations, no other properties are assumed. Conversions are applied with Nuprl's cut rule: If c is a conversion, and $c(p)(t)$ returns the triple $<R, t', tac>$, then $t\ R\ t'$ may be proved with tac and used as an assumption.

In the remainder of this section we describe how conversions are constructed and composed. Our approach provides operators that construct conversions using primitive inference rules and lemmas and provides combinators that build conversions from simpler ones. This modular higher-order approach to rewriting originated with Paulson who provides an account in [16]. (See also Nipkow for a different account based on higher-order unification [15].) Rather than duplicate parts of Paulson's account, we shall instead focus on what is novel about our approach: how we coordinate rewrites over arbitrary relations and our use of lemmas to direct inference and validation in a theory with undecidable typing problems.

3.1 Basic Conversions

Basic conversions are of two types: primitive and lemma-based. Primitive conversions, such as the identity conversion *IdConv* and (values of) *IdConvWithR*, prove their rewrites valid with primitive inference rules. *IdConvWithR* takes a relation R as an argument and returns the conversion $\lambda p.\lambda t.<R, t, tac>$. A Nuprl *library*, an ordered collection of definitions, theorems, and other objects, is used to construct a database of lemmas expressing relational properties. For the identity rewrite, *IdConvWithR* searches this database for a lemma of the form

$$\vdash t\, R\, t\,.$$

If no such lemma is found, the relation is assumed irreflexive and the conversion fails. Otherwise the conversion succeeds and *tac* proves $t\ R\ t$ with the found lemma. *IdConv* is similar to *IdConvWithR*. But instead of taking a relational argument, type inference routines described in [8] infer a T for the relation $=_T$. The conversion fails when T cannot be inferred, but this has not been a problem in our experience.

Other primitive conversions include *FailConv*, the always failing conversion, *ReduceConv*, which replaces redices by their contracta, and *Simplify* which performs arithmetic simplification.

Lemma-driven conversions construct rewrite triples from lemmas in a user's library. These lemmas take the form

$$\forall x_1 : T_1.\, \forall x_2 : T_2. \ldots \forall x_n : T_n.\, A_1 \Rightarrow A_2 \Rightarrow \ldots \Rightarrow A_m \Rightarrow s\ R\ s'\,.$$

The function *LemmaToConv* takes two arguments, a lemma l, in which m, the number of assumptions, is zero, and a relation R, and returns $\lambda p.\lambda t.{<}R, t', tac{>}$. *LemmaToConv* matches t against the left operand of the relation $s\ R\ s'$[3] in l. If the match fails, the conversion fails. Otherwise, the variables $x_1, \ldots, x_n$ are bound by the match and used to instantiate s' which is returned as t'. When executed the tactic *tac* applies l to prove $t\ R\ t'$.

Conditional rewrites are constructed similarly. *ImpLemmaToConv* takes as inputs a lemma l, which may have assumptions, a relation R, and a tactic. Before constructing its rewrite triple, *ImpLemmaToConv* applies the input tactic to the instantiated assumptions $A_1, \ldots, A_m$ and fails if this application fails. *Tac* appropriately combines both the lemma l and the input tactic.

Both *LemmaToConv* and *ImpLemmaToConv* are driven by first-order matching. We have also implemented powerful versions that are based on second-order matching. As second-order matches are not necessarily unique, these functions take an additional argument, a match discriminator function that chooses an appropriate match from a set of matches.

3.2 Composing Conversions

Given functions that construct basic conversions, the next step is to provide operators that combine them. The two basic combinators are *ORELSEC* and *THENC*. The former provides selective composition and the latter a method of sequentially composing conversions.

ORELSEC is based on ML failure. In ML, the expression e_1 ? e_2 computes the value of e_1 and if that fails it computes the value of e_2. Thus, we define $(c_1\ \mathit{ORELSEC}\, c_2)(p)(t)$ as

$$c_1(p)(t)\ ?\ c_2(p)(t)\,.$$

THENC uses a generalized kind of transitivity reasoning to sequentially compose conversions. Unlike selective composition, this requires use of the lemma database. Given conversions c_1 and c_2, $(c_1\ \mathit{THENC}\, c_2)(p)(t)$ computes the triples

$$\begin{aligned} {<}R_1, t_1, tac_1{>} &= c_1(p)(t) \qquad \text{and} \\ {<}R_2, t_2, tac_2{>} &= c_2(p)(t_1)\,. \end{aligned}$$

[3]There are functions *RevLemmaToConv* and *RevImpLemmaToConv* which match against the right operand.

If either conversion fails, then *THENC* fails. Otherwise, *THENC* uses R_1 and R_2 as keys and searches the database for a sequencing lemma of the form

$$t\ R_1\ s,\ s\ R_2\ r\ \vdash\ t\ R_3\ r\,.$$

If such a lemma is found, the triple $<R_3, t_2, tac>$ is returned, where tac_1, tac_2, and the lemma are appropriately combined into tac.

There is no need for the relations to be identical. For example,

$$t < s,\ s \leq r\ \vdash\ t < r$$

is a valid sequencing lemma. Our implementation insists that there is at most one sequencing lemma for any pair of relations; more general approaches allowing multiple sequencing lemmas are possible. Such approaches would be implemented analogously to the congruence reasoning routines described in Section 3.3.2.

ORELSEC and *THENC*, along with *FailConv* and *IdConv*, provide the basis for multi-way choice and repetition. The operator *FirstC* returns the first successful conversion from its argument list $[c_1; ...; c_n]$. It is equivalent to

$$c_1\ \mathit{ORELSEC} \ldots\ \mathit{ORELSEC}\, c_n$$

and is defined recursively in terms of *ORELSEC* and *FailConv*. *RepeatC*(c) repeatedly applies the conversion c until failure. It is defined recursively as

$$(c\ \mathit{THENC}\,(\mathit{RepeatC}(c)))\ \mathit{ORELSEC}\ \mathit{IdConv}\,.$$

With *FirstC* and *RepeatC* it is easy to construct a general chaining procedure where relational inferences are combined to the extent justifications are provided in the database.

3.3 Congruence Conversions

3.3.1 Congruence Proofs

Subterm rewriting requires the construction of tactics that generate congruence proofs. These proofs can be subtle and in practice more difficult to construct than reflexivity and transitivity proofs. We shall first examine these proofs in the simplest possible setting, equality congruence, and then consider general relational congruence.

Given $t =_T t'$, to conclude $r[t/x] =_{T'} r[t'/x]$, we must prove that equal members x in T yield equal members r in T' (x may be free in r), i.e.,

$$\frac{\vdash t =_T t' \qquad x\!:\!T \vdash r \in T'}{\vdash r[t/x] =_{T'} r[t'/x]}\,.$$

The extra premise, that r is functional in x, takes the form of a membership goal and, as discussed in Section 2, is in general undecidable.

Congruence proofs are constructed by induction on the structure of Nuprl terms. A term is a tree. It is specified by an n-ary term constructor and its n subterms. If the subterms t_i of a term T are rewritten via equality to t'_i, congruence is proven by decomposing T into its immediate subterms and proving that T is functional in the types of these terms. The t_i are then either proved equal to t'_i by reflexivity (*equality*), proved equal by assumption (*hypothesis*), or recursively decomposed. We illustrate this with the following example. Given $r =_A s$, we prove

$$\vdash\ f(g(r)) =_C f(g(s))$$

under the assumptions $f:B \to C$, and $g:A \to B$ as follows.

1	$\ldots \vdash f(g(r)) =_C f(g(s))$	*by intro using* $B \to C$
1.1	$\ldots \vdash f =_{B \to C} f$	*by equality*
1.2	$\ldots \vdash g(r) =_B g(s)$	*by intro using* $A \to B$
1.2.1	$\ldots \vdash g =_{A \to B} g$	*by equality*
1.2.2	$\ldots \vdash r =_A s$	*by hypothesis*

Here we have linearized the proof tree, i.e., 1.1 and 1.2 are the subgoals arising from 1, and 1.2 in turn has two further subgoals. In the example, observe that each step in the congruence proof can be viewed as justifying a rewrite with respect to some (different) equality relation. As we "peel" through the various term constructors, our rewrite relation changes.

This approach of proving functionality by recursive decomposition generalizes to congruence proofs for arbitrary relational rewrites. Suppose T is a term containing $n \geq 1$ subterms t_i that are rewritten with respect to n (possibly distinct) relations. Viewing T as a tree, for each term constructor θ in the path from each t_i to the root of T we must prove that θ is functional in the relations used to rewrite its immediate subterms. Formally, θ is said to be R_0-functional in the relations $R_1, \ldots, R_n$ whenever

$$\theta(t_1, \ldots, t_n)\ R_0\ \theta(t'_1, \ldots, t'_n) \tag{1}$$

is provable under the assumptions

$$\begin{array}{ccc} t_1 & R_1 & t'_1, \\ & \vdots & \\ t_n & R_n & t'_n \\ C_1 & & \\ & \vdots & \\ C_m. & & \end{array} \tag{2}$$

The $C_1, \ldots, C_m$ are additional auxiliary conditions which may be required of the t_i. We call the lemma that proves that (1) follows from the assumptions in (2), a *congruence lemma*. A congruence proof consists of recursively decomposing a term by applying congruence lemmas until the relations among the resulting subterms follow by reflexivity or from a given conversion.

As a simple example, suppose we are given the formula $\forall x_1 : T_1. \ldots \forall x_n : T_n. T$ and the conversion c where $c(p)(T) = \langle \Leftrightarrow, T', tac_1 \rangle$. Then n applications of the congruence lemma

$$\begin{array}{ll} \vdash \forall T{:}U_1. \forall P, Q{:}T \to U_1. & (\forall x{:}T. (P(x) \Leftrightarrow Q(x))) \\ \Rightarrow & (\forall x{:}T. P(x) \Leftrightarrow \forall x{:}T. Q(x)), \end{array}$$

which proves that the relation $\forall$ is $\Leftrightarrow$-functional in the relation $\Leftrightarrow$, reduces proving

$$\forall x_1{:}T_1. \ldots \forall x_n{:}T_n. T \Leftrightarrow \forall x_1{:}T_1. \ldots \forall x_n{:}T_n. T'$$

to proving $T \Leftrightarrow T'$ (in the context of the $x_i{:}T_i$), which is proved by the tac_1.

3.3.2 Inferring Rewrite Relations

In our implementation, the function *SubConv* provides the basis for subterm rewriting. *SubConv*(c) is a conversion that applies the conversion c to the immediate subterms of a term. The tactic it produces justifies the subterm rewrite. Repeated application of *SubConv* allows rewriting of arbitrary subterms.

To construct a rewrite triple, *SubConv* use the lemma database as a source for congruence lemmas. However, there may be more than one choice of R_0 for a given θ and $R_1, ..., R_n$. For example, suppose that θ is the multiplication operator $*$, and R_1 and R_2 are $=$ and $<$. Then with an additional inequality condition, the rewrite relation R_0 may be either $<$ or $\neq$. That is, both

$$a = c,\ b < d,\ a > 0 \vdash a * b < c * d \tag{3}$$

and

$$a = c,\ b < d,\ a > 0 \vdash a * b \neq c * d \tag{4}$$

are valid congruence lemmas. Moreover, in the proper proof context, each of these lemmas could find application.

How one determines which congruence lemma is applicable is an important design decision. If *SubConv* chooses an improper rewrite relation, then a rewrite may either eventually fail, as it will be unable to create a tactic that constructs a congruence proof, or the relation chosen may be too weak, rendering the rewrite useless. Hence, it is important to have an effective strategy for selecting rewrite relations. In this section we outline two methods for controlling relational inference: a powerful but computationally expensive method, and a simplified heuristic method that works well in practice. Our implementation is based on the latter.

The most general method of subterm rewriting is to construct all possible rewrites as permitted by the congruence lemmas contained in the database. Such a method necessitates a generalized type of conversion that returns not a single rewrite triple, but rather a set of triples. In such a setting, if a term t's outer most operator is the n-ary constructor θ, then $SubConv(c)(p)(t)$ would:

1. Associate with each of the n immediate subterms t_i of t either the set of rewrite triples returned by $c(p)(t_i)$, or if this fails, the singleton triple returned by $IdConv(p)(t_i)$. If all n subterm rewrites fail, then *SubConv* fails.

2. Form all possible n-tuples where the ith element of the tuple comes from the triple set associated with the term t_i. For each such tuple, search the lemma database for a congruence lemma which states that for some R_0, θ is R_0-functional in the tuple's n rewrite relations. If such a lemma exists, use it to construct a new rewrite $<R_0, \theta(t'_1, ..., t'_n), tac>$ where tac applies l and the tac_i.

3. Return the set of new triples or fail if the set is empty.

The effect of the above construction is that *SubConv* generates, bottom-up, all possible congruence proofs. The resulting set of triples can be used to selectively add new facts to the hypothesis list of the sequent and for subsequent inference.

While this approach makes the fullest use of the lemma database, its time complexity can be exponential in the depth of the rewritten subterms. Our solution to this combinatorial explosion rests on the observation that in most cases, when there is a choice among rewrite relations, it suffices to pick the strongest relation. For example, it is preferable to know that two types are equal instead related by bi-implication (if and only if). Similarly, bi-implication is a stronger relation than implication, and less-than is stronger than less-than-or-equal. So, for example, one generally prefers to use the congruence lemma given by (3) over (4).

Our approach, which is linear in the depth of the rewritten subterms, always returns the strongest possible rewrite relation. We use a user provided table to determine relative relational strength. $SubConv(c)(p)(t)$ produces a single rewrite triple as follows:

1. The function $c(p)$ is applied to each subterm t_i. For each subterm this yields the triple $<R_i, t'_i, tac_i>$, or, failing that, the triple $IdConv(p)(t_i)$. If all n conversions fail, then *SubConv* fails.

2. The operator θ and the relations R_i are used to index into the library for congruence lemmas that specify relations R_0 such that θ is R_0-functional in the R_i. When more than one such lemma is found, the one with the strongest R_0 is chosen. If no such lemma is found, *SubConv* fails.

3. A tactic *tac* is constructed that applies the lemma found in the previous step. This reduces the proof of $t \ R_0 \ t'$ (where $t' = \theta(t'_1, ..., t'_n)$) to the subgoals $t_i \ R_i \ t'_i$. *Tac* then proves each such subgoal by tac_i.

4. The rewrite triple $<R_0, t', tac>$ is returned.

3.3.3 Subterm Traversal

SubConv provides the basis for traversing terms recursively. It is now easy to write an operator $Depth(c)$ that recursively rewrites all subterms of a term in depth-first order. Its recursive definition is

$$((SubConv(Depth(c))) \ THENC \ (RepeatC(c))) \ ORELSEC \ (RepeatC(c)) .$$

Similarly, top-down rewriting is accomplished by $Top(c)$ whose recursive definition is

$$(ProgressC(RepeatC(c))) \ ORELSEC \ (SubConv(Top(c))) .$$

(Note that $ProgressC(c)$ is a combinator that fails when c behaves like *IdConv*.)

3.3.4 An Example

Let c be a conversion that rewrites t to t' under the less-than relation $<$. Suppose m is non-negative, the sequent p contains appropriate well-formedness hypotheses, and the database contains the congruence lemmas given by

$$a = c, \ b \leq d \vdash a < b \Rightarrow c < d \tag{5}$$

and (3). Then

$$Top(c)(p)(n < m * t)$$

returns the triple

$$<\Rightarrow, n < m * t', tac>. \tag{6}$$

Tracing *Top*'s recursive execution, we find that after two calls to *SubConv* $c(p)(t)$ returns the triple $<<, t', tac_1>$. The second *SubConv* uses (3) and the previous triple (as well as the triple where m is rewritten by the identity conversion) and returns the triple $<<, m * t', tac_2>$. Finally, the first call to *SubConv* uses (5) and returns the final triple given in (6).

The end result of the above rewrite is that $n < m * t \Rightarrow n < m * t'$ may be added to the hypothesis list and this new hypothesis can be used for other rewrites or forward and backchain deductions. This example demonstrates how *SubConv* uses the lemma database to reason about inequalities. It also illustrates how our approach is strictly stronger than Paulson's.

4 Conclusion

Our system is a practical solution to an important theorem proving problem. It dramatically raises the level of user/system interaction; moreover, it provides a foundation for building high-level proof procedures such as decision procedures based on equational reasoning and term normalization. Examples are given in [2, 3].

One area for improvement and future research is rewrite efficiency. Our approach essentially justifies rewrites twice: once when using the lemma database to construct rewrites, and again when the tactics are executed. An alternative is to build proofs directly; however, this approach is wasteful when conversions fail and *ORELSEC* uses failure as selective composition. Another possibility is to reflect Nuprl's metalanguage into the object language and prove conversions correct. Such an approach would obviate the need to construct tactics as the rewrite is formally proved valid. Howe has had some success building a partial-reflection library and verifying basic rewrite strategies [9]. Unfortunately his library is limited in scope and has its own efficiency problems which stem from inefficiencies in Nuprl's evaluator. There is a research effort at Cornell to design a reflected object language that will better support this style of rewriting [1].

References

[1] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Symposium on Logic in Computer Science*. Computer Society Press of the IEEE, 1990.

[2] David Basin. A term equality problem equivalent to graph isomorphism. *Information Processing Letters*, 51:61 – 66, 1994.

[3] David Basin and Peter DelVecchio. Verification of combinational logic in Nuprl. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Ithaca, New York, 1989. Springer-Verlag.

[4] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[5] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, pages 95–120, 1988.

[6] R.J. Cunningham and A.J.J Dick. Rewrite systems on a lattice of types. *Acta Informatica 22*, pages 149–169, 1985.

[7] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[8] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.

[9] Douglas J. Howe. Computational metatheory in Nuprl. In *9th International Conference On Automated Deduction*, pages 238–257, Argonne, Illinois, 1988.

[10] Douglas J. Howe and Evan Steeg. Rewrite.ml. Nuprl Internal Document, 1985.

[11] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, pages 797–821, October 1980.

[12] Gérard Huet. Deduction and computation. In *Fundamentals of Artificial Intelligence*, pages 39–74. Springer-Verlag, 1986.

[13] Paul Jacquet. Program synthesis by completion with dependent subtypes. In *9th International Conference On Automated Deduction*, pages 550–562, Argonne, Illinois, 1988.

[14] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[15] Tobias Nipkow. Equational reasoning in Isabelle. *Science of Commputer Programming*, 12:123–149, 1989.

[16] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[17] G. Smolka, W. Nutt, J. Meseguer, and J.A. Goguen. Order-sorted equational computation. In *Colloquium on the Resolution of Equations in Algebraic Structures*, Lakeway, Texas, May 4-6 1987.