Vision for the Future of Logic Programming: Laying the Foundations of a Modern Successor to Prolog. A Workshop in Association with the International Logic Programming Symposium. Portland, OR, Dec. 1995, pages 37–46.

Gödel with User-defined Evaluable Functions

Sergio Antoy *David Shapiro* *Janet Vorvick*

Portland State University

Abstract

We report our experience extending a logic programming language with a first-order functional component. Functional expressions are evaluated by narrowing and can contain logic variables for a seamless integration with the logic component of the language. Our experiment shows that our extension is achievable with a modest effort. It also suggests some changes to the syntax and semantics of the initial logic component of the language that would ease the use and implementation of the functional component.

1 Introduction

Logic languages may benefit from allowing the definition of functions in addition to predicates and performing functional computations in addition to logic computations. A significant problem presented by the presence of functions in a logic language is this: what shall we do when a function is applied to arguments that are (or contain) logic variables. A solution to this problem is narrowing.

The focus of this note is on narrowing as the means for the integration of functional and logic computations. Our specific goals are the application of *needed narrowing* and the implementation of narrowing in Prolog. According to a recent survey [8], narrowing is the most common operational principle for the integration of the functional and logic programming paradigms. Needed narrowing [2] is a narrowing strategy that computes only independent substitutions and minimizes the number of steps of a derivation when the common subterms of a term are shared.

Many interesting functional logic languages which adopt narrowing as their operational principle have been designed in the last decade, e.g., *ALF* [7], *BABEL* [13], *K-LEAF* [6], and *LPG* [4]. These neither make use of needed narrowing, due to its novelty, nor use Prolog as underlying run-time environment.

This work has been supported in part by the National Science Foundation under grant CCR-9406751.

Authors' addresses: Department of Computer Science, Portland State University, P.O. Box 751, Portland, OR 97207, U.S.A., {antoy,davids,janetv}@cs.pdx.edu.

Since the design and implementation of a new language is a major undertaking, our approach has been to contain the effort of producing a functional logic language by extending with a functional component an already implemented logic language. Our approach, despite having some drawbacks, not only simplifies the implementation, but also gives some insight into the design of logic languages. In particular, we explore the cost of building a functional component on top of a logic one, and explore some changes in design and semantics suggested by the presence of functions.

The logic language that we chose for our extension is Gödel [10]. Our choice was motivated by the fact that Gödel has a mostly pure declarative logic, it is strongly typed, and it has a module system and a rich set of logic expressions. An implementation of Gödel is publicly available for many common platforms. The source code of the compiler comes with the distribution and the compiler translates Gödel programs into source Prolog for the popular SICStus compiler.

The remainder of this paper is organized as follows. In Section 2 we recall narrowing, which is at the core of our extension. In Section 3 we present the syntax of our extension. In Section 4 we describe the operational semantics of our extension. In Section 5 we outline the major changes and additions that were required by our implementation. In Section 6 we show, with reference to examples, some advantages resulting from the addition of a functional component to Gödel. In Section 7 we draw some conclusions from our experience.

2 Narrowing

Most proposals for the integration of functional and logic programming languages having a sound and complete operational semantics are based on narrowing [8]. In this section we give a very informal account of this concept.

A rewrite system, see [5, 11] for excellent tutorials, is a set of rewrite rules that tell us how to compute. For example, the following rules describe how to perform additions on natural numbers, which are represented by terms built with 0 and S:

$$\begin{array}{rcll} 0+x & \rightarrow & x & \mathrm{R}_1 \\ S(x)+y & \rightarrow & S(x+y) & \mathrm{R}_2 \end{array}$$

Computations are applications of equational reasoning. The symbol "$\rightarrow$" is intended as an *oriented* equality relation. A computation, referred to as *rewriting*, is a sequence of steps in which an instance of a rule's left hand side is replaced by the instance of the corresponding right hand side. For example, to rewrite $S(0) + S(0)$ we apply rule R_2 to the whole term by instantiating x to 0 and y to $S(0)$. This step yields $S(0 + S(0))$. Now we apply rule R_1 to the subterm $0 + S(0)$ by instantiating x to $S(0)$. This step yields $S(S(0))$. Since no rule is applicable to this term, we regard it as the result of the computation.

Narrowing generalizes rewriting by computing with partial information.[1] A narrowing step can be seen as a two-phase process. During the first phase a non-variable subterm s of the term t which is the object of the computation is unified with the left hand side of a rewrite rule $R = l \rightarrow r$. The result is a unifier σ such that $\sigma(l) = \sigma(s)$. During the second phase σ is applied to t and the rewrite rule R is fired to replace, in $\sigma(t)$, $\sigma(s)$ with $\sigma(r)$. For example, to narrow $z + S(0)$ we first unify the whole term with the left hand side of R_2, which yields the unifier $\{z \mapsto S(x), y \mapsto S(0)\}$ and

[1]The missing information is "created by an educated guess" that makes it possible to continue a computation that otherwise would stop.

then fire R_2 on $S(x) + S(0)$, which yields $S(x + S(0))$. Here, we instantiated not only the variables of the rules, but also the variables of the term which is the object of the computation, e.g., z.

Narrowing is a non-deterministic operation. Every step entails the choice of both a position and a unifier (for the systems we are interested in, the choice of a rule is implied by the choices of a position and a unifier). These choices are the responsibility of a *strategy* whose main goal is the elimination of useless choices. *Needed* narrowing [2] is a strategy that performs only steps that are, in a precise technical sense, needed to compute a solution of an equation. On ground terms it performs what is referred to as lazy evaluation in functional programming. This strategy is sound, complete, and optimal for *inductively sequential* rewrite systems [1], a class that encompasses the first-order programs of functional programming languages such as *ML* and *Haskell*.

3 Syntax

In this section, we discuss the syntax of our language. We assume some familiarity with the Gödel programming language and we describe only the syntax of our extensions. We use the word "syntax" in a broad sense to indicate what is or is not legal in our extension. For example, the correct typing or the left linearity of rewrite rules are not described formally by a grammar, but are included in this section.

A design goal of our extension is maximum compatibility with Gödel. Thus, a syntactically correct program of our extension that does not make use of user-defined evaluable functions is also a syntactically correct Gödel program. The converse is also true, with a small exception. A syntactically correct Gödel program is syntactically correct for our extension, too, as long as it does not declare two symbols that are predefined in our extension, but not in Gödel. In the following discussion, a *term* is either the right hand side of a rewrite rule or an argument of a predicate.

Next we present the changes that our extension makes to Gödel. Later we motivate our design decisions.

1. The symbol `=>` is a reserved, binary, infix, overloaded operator used for the definition of evaluable functions. Its left and right operands are interpreted as the left and right hand sides of a rewrite rule. A rewrite rule has the form l `=>` r `<-` c. The condition c is a possibly empty conjunction of equations such as those that can be found in the body of a clause.

2. Rewrite rules must satisfy all the conditions imposed on a left-linear, polymorphic, many-sorted, constructor-based term rewriting system [5, 11]. Extra variables in the right side and/or condition of a rule are allowed if they satisfy the confluence criteria established in [15]. A convention, which we will discuss in the next section, on the order of presentation of the rules makes our rewrite systems confluent and suitable for needed narrowing.

3. A function is *evaluable* when there exists in a program a rewrite rule defining it, otherwise it is a *data constructor*. A data constructor may be declared only in the same module that declares the type of its range. By contrast, an evaluable function may be declared in any module and may have any type as its range, including types defined in the system modules. A rewrite rule defining an evaluable function may occur only in the module declaring that function.

4. Our extension predefines the type `Boolean` and changes the category of the symbols `True` and `False` from proposition to constant data constructor of this type. Boolean functions can

be defined by the user and do not differ from any other user-defined evaluable function. In particular, predefined Boolean operators, such as `&` and `\/`, can appear in Boolean terms. Likewise, Boolean terms, including the values `True` and `False`, can appear as atoms in the body of a clause.

5. A conditional expression has the form `IF` c `THEN` e_1 `ELSE` e_2, where c is a Boolean expression and e_1 and e_2 are expressions of the same type. A conditional expression can be a subterm of a term. The `ELSE` branch of the conditional expression is mandatory. The value of the conditional expression is the value of either e_1 or e_2 according to the truth of the condition c.

Gödel programs are accepted by our extension if they do not declare the operator `=>` or the identifier `Boolean`. Gödel programs executed by our extension give the same results as Gödel.

Point number 3 preserves condition M4 of the Gödel's module system for non-evaluable functions, but refines it for evaluable functions. A violation of the condition on data constructors would allow a module to extend a type defined by another module. A violation of the condition on evaluable functions would allow a module to change a data constructor declared by another module into an evaluable function. In both cases, the possibly inaccessible definitions of existing predicates and/or functions would be invalidated. It is interesting to observe that point 3 of our syntax is the union of condition M4 of the Gödel and condition M4 of the Escher programming languages [12].

Boolean functions are predicates. The convenience of having both Boolean functions and predicates will be addressed in the next section.

As in Gödel, there is no `DELAY` declaration for functions. We will address in our conclusions pros and cons of this choice.

4 Operational Semantics

The operational semantics of our extension is almost identical to that of Gödel. Some Gödel system modules already provide some evaluable functions, such as arithmetic operations on numbers, concatenation of strings, etc. Users, however, cannot define evaluable functions. Our extension simply gives users this possibility. A function defined in a system module instantiates any uninstantiated arguments to which it is applied; thus Gödel evaluates functional expressions by narrowing. The only difference between functions defined in the system modules and functions defined via our extension is in the evaluation strategy. We perform needed narrowing, whereas Gödel adopts innermost narrowing.

If an expression e containing evaluable functions is the argument of a predicate P, then before the invocation of P, e is non-deterministically narrowed until an expression not containing evaluable functions is computed. This approach implies that equations are valid if and only if their operands can be narrowed to the same constructor term. This kind of equality, known as *strict*, is commonly adopted for non-terminating rewrite systems.

In most functional languages based on pattern matching, such as *ML* and *Haskell*, the first rule that matches an expression is applied to it. This implies that some patterns in some rules following the first one are more specialized than they appear. For example, referring to the rewrite system of Section 2, consider the definition of the function *IsZero*.

$$\begin{array}{lcl} \mathit{IsZero}(0) & \rightarrow & \mathit{True} \\ \mathit{IsZero}(\_) & \rightarrow & \mathit{False} \end{array}$$

The pattern '_' of the second rule only matches terms of the form $S(\_)$. In a narrowing computation both rules of *IsZero* may be applied to the same term. A literal interpretation of the patterns would create an obvious inconsistency as, e.g., $IsZero(t)$, where t is an uninstantiated variable, would be narrowed to both *True* and *False* with non-independent unifiers. Since the familiar functional interpretation of patterns in the rules defining a function simplifies definitions, as shown by our example, we preserve it in our extension. Thus, the pattern '_' of the second rule is interpreted as $S(\_)$, i.e., the complement of 0. This convention has the advantage of producing only rewrite systems that are confluent and are in the domain of application of needed narrowing.

The usefulness of having both predicates and functions in the same language stems from the opportunity to perform less general, but more efficient, computations. Function definitions interpreted with our convention on patterns make rewrite systems inductively sequential [1]. Inductive sequentiality implies that in any term t that is not fully evaluated there is a position that *must* eventually be narrowed to evaluate t. This position is called *needed*. A needed position of a term can be easily computed by unification and is the key for the implementation of needed narrowing, which is more efficient than other lazy narrowing strategies. Predicates can be used to define computations in which needed positions may not exists. An example of such a predicate is the so-called *parallel-or* shown next by both rewrite rules and Gödel code.

$$Or(True, \_) \to True \qquad\qquad \texttt{Or(True,\_).}$$
$$Or(\_, True) \to True \qquad\qquad \texttt{Or(\_,True).}$$

Gödel makes no convention on the order of presentation of the clauses of a predicate. As a consequence, there may be no needed position in a term of the form `Or(`t_1`,`t_2`)`. This expression evaluates to `True` as long as one argument evaluates to `True`, even if the other argument is undefined. Although the above predicate is more general than an inductively sequential function computing Boolean disjunction, its generality is computationally much more expensive to achieve.

Narrowing could allow the execution of some programs that are not legal for Gödel. We will propose in our conclusions a semantic change to Gödel that yields a conservative extension of the class of programs that may be executed.

5 Implementation

Our implementation can be described as a set of changes to the Gödel parser and a set of additions to the Gödel code generator.

We take advantage of the fact that narrowing is a binary relation on terms and thus narrowing can be described naturally by a predicate. We predefine the operator `=>` as a predicate. Very few additional changes are required to make the parser accept and process rewrite rules. Some effort is necessary for point 3, since the declaration of a function F gives no indication of whether F is evaluable. Hence we must either use some kind of look-ahead approach or do two passes over the code to determine the presence or absence of rewrite rules defining F. A declaration that does not discriminate between data constructors and evaluable functions creates other problems discussed next. We will propose in our conclusion a syntactic change to Gödel for the solution of this problem.

The clauses defining `=>`, i.e., the rewrite rules, are passed to an extension of the code generator whose purpose is the generation of Prolog code that implements narrowing. The compilation process in Gödel makes use of auxiliary files for recording and retrieving internal objects. Our extension needs one extra file to distinguish data constructors from evaluable functions. This file is loaded in the run-time environment, since this information is necessary for processing goals.

The current implementation of our extension provides a large subset of the language described in the previous sections. The features currently omitted from the system, with the exception of the narrowing strategy, can easily be implemented at the user level. The narrowing strategy implemented at the time of this writing is leftmost innermost rather than needed. Leftmost innermost narrowing is much simpler than needed narrowing and we are currently assessing the efficiency of various alternatives for the implementation of needed narrowing in Prolog [3]. The implementation of needed narrowing is in progress.

6 Examples

We compare the logic and the functional-logic versions of a same program. Following a well-established tradition in logic programming, our first example deals with family relations. We present only a fragment of the program. In Gödel the program fragment is

```
PREDICATE Father, Mother, Parent, PaternalGrandfather : People * People.

Father(Joe,Tom).
...
PaternalGrandfather(x,y) <- SOME [z] (Father(x,z) & Father(z,y)).
Parent(x,y) <- Father(x,y) \/ Mother(x,y).
```

where the type People and the predicate Mother are suitably defined. In our extension, the same program fragment could be coded as

```
FUNCTION  Father, Mother, PaternalGrandfather : People -> People.
PREDICATE Parent : People * People.

Father(Joe) => Tom.
...
PaternalGrandfather(x) => Father(Father(x)).
Parent(x,Father(x)).
Parent(x,Mother(x)).
```

In the functional-logic version, the definition of Father makes clear that Tom is the father of Joe rather than the reverse. During the execution of the program no choice points need to be created for the computations of Father and PaternalGrandfather on ground arguments, since these relations are functional. An implementation taking advantage of this fact could run faster. The relation Parent is still expressed by a predicate, since there are no functional dependencies between its arguments. The definitions of PaternalGrandfather and Parent avoid extraneous variables, a quantifier and an operator, and trade a clause with a relatively complicated body for two simpler unit clauses. In this small example, the advantage of the availability of functions is a more natural, terser, and more easily understood program.

This example also shows advantages with respect to functional programming. In a purely functional version of the program, Parent would have to be a function that returns a pair or a set of People. Furthermore, the functional program could also require an inverse function of Father, which would have to return a set or a list of `People`. In both cases, the programmer has to write additional code and use more complex types.

A substantial advantage of having functions stems from the ability of needed narrowing to perform only needed computations. For example, logic programming is an ideal paradigm for the class of algorithms known as generate-and-test. A naive implementation of these algorithms in logic is very easy though inefficient. A standard technique to achieve efficiency is "…to 'push' the tester inside the generator as deeply as possible." [14, p. 252], which complicates coding, limits code reusability, and obfuscates algorithms. Needed narrowing joins all the simplicity, reusability, and clarity of the naive implementation with the efficiency of a tester which is completely inside the generator. We discuss these issues in detail in our next example.

The example originates from the program for the wolf, goat, and cabbage puzzle proposed in [10, p. 131]. Our computation is in two steps: first we *potentially* generate the search space for the *whole* puzzle and then we look for a goal in the search space. In general, the search space of a generate-and-test problem can be huge or infinite, however, only the portions of the space that are needed by the tester end up being computed by the generator. The following code fragment abstractly defines a structure for the representation of the search space and a function for the computation of the portion of search space reachable from a state.

```
CONSTRUCTOR SpaceRep / 2.
FUNCTION    SpaceRep : a * b * Set(SpaceRep(a,b)) -> SpaceRep(a,b).

FUNCTION    Space : State * List(Move) * List(State)
                  -> Set(SpaceRep(State,List(Move))).

Space(state, moves, history) =>
     { SpaceRep(new,[move|moves],Space(new,[move|moves],[new|history]))
         : Move(state, history, move, new) }.
```

The right hand side of the rule of `Space` is slightly more complex in the actual code, due to Gödel's scoping rules for intensional sets.

The type representing the search space, introduced by the Gödel category `CONSTRUCTOR`, has only one data constructor, introduced by the Gödel category `FUNCTION`. Following a common practice in functional programming we overload the symbol `SpaceRep` to identify both the type and its data constructor.

A point of the search space consists of a state s of the problem, the sequence m of moves performed to compute s, and the search space r below s. For efficiency, r does not represents states that are ancestors of s, i.e., it avoids loops in its paths. The function `Space` takes three arguments: a state s, the sequence m of moves to compute s, and the sequence h of ancestors of s. When applied to s, m, and h, `Space` computes r and thus easily constructs the whole search space of a problem as shown later. The argument h of `Space` is used only to avoid loops in the paths of the search space and is unnecessary for terminating problems. For the wolf, goat, and cabbage puzzle we define

```
Move(current_state, history, move, new_state)
    <- Applicable(move, current_state)
     & ApplyMove(move, current_state, new_state)
     & Legal(new_state)
     & NoLoops(new_state, history).
```

where the predicates `Applicable`, `ApplyMove`, `Legal`, and `NoLoops` as well as the types `State` and `Move` are already defined in [10, p. 133–134].

An advantage of this formulation is that the generator is logically independent and textually separated from the tester. One can use different testers with the same generator for implementing different search strategies. For example, instead of the tester presented in [10, p. 131], which implements depth-first search, we show a function that may look for a goal using a variety of strategies including, but not limited to, depth-first.

```
PREDICATE FindGoal : List(SpaceRep(State,List(Move))) -> List(Move).

FindGoal([SpaceRep(state,moves,children)|open]) =>
    IF Final(state) THEN moves
    ELSE IF children=Null THEN FindGoal(open)
    ELSE FindGoal(Join(children,open)).
```

The predicate `Final` is defined in [10, p. 133]. The function `Join`, whose type is `Set(a) * List(a) -> List(a)`, builds a list containing the elements of both its arguments. The order of the elements in the result establishes the search strategy. For example, placing the elements of `children` at the end of the list of points of the `open` space yields the breadth-first strategy, whereas placing the elements at the front yields the depth-first strategy. The moves to find a goal from an initial state `s` are computed by

```
FindGoal([SpaceRep(s,[],Space(s,[],[s]))])
```

We reiterate that the generator computes only the points of the search space visited by the tester. A predicate to compute the search space could be obtained from the function `Space` with trivial syntactic transformations. Its semantics, though, would be quite different—upon being called, the predicate would compute, or attempt to, the whole search space regardless of the points that would be later visited by the tester.

7 Conclusions

Gödel is a successor to Prolog as the extensive comparison of the two languages in [10] indicates. We believe that allowing user-defined evaluable functions (even when this feature is limited to first-order) is advantageous in terms of expressiveness and performance. Our work shows that adding a functional component to Gödel requires a modest effort, does not require major changes to the syntax and semantics of the language, and highlights a few interesting issues that we discuss next.

The syntax of a language should make it apparent whether or not a function is evaluable *at the time of its declaration*, rather than by presence or absence of rewrite rules defining it, which may reside in a file different from that of the function declaration. This would also ease modular programming in Gödel by allowing the compilation of a module M from the `EXPORT` part of each module N on which M depends regardless of the content or even the existence of N's `LOCAL` part. This is impossible with the current syntax, since the rewrite rules defining a function may reside in the `LOCAL` part of a module.

Declaring the data constructors of a type contextually with the type itself, as is done in some functional programming languages, would solve the problem. For example, a polymorphic type

`Tree` and its data constructors `Leaf` and `Branch` could be more compactly declared as follows

```
TYPE Tree (a) = Leaf | Branch (a, Tree (a), Tree (a)).
```

Using the full potential of narrowing would allow some non-trivial extensions to the semantics of Gödel. For example, suppose that a predicate `P` is defined by

```
P(x,y,z) <- IF x=y THEN Peq(z) ELSE Pne(z).
```

for some suitable predicates `Peq` and `Pne`. If when `P` is called its first and/or second arguments are non-ground, the execution of the call, according to the semantics of Gödel, flounders. However, this need not be the case with narrowing. During the execution of the body of `P`, the expression `x` = `y` could be narrowed to either `True` or `False`, even when it contains variables. Any variable of `x` and/or `y` would be instantiated, if necessary for the evaluation of the condition, before the execution of either branches of the conditional. We conjecture that this behavior would conservatively extend the set of executable programs. Similarly, the computation of an intensional set may flounder, due to the presence of variables in the elements of the set. Narrowing has the potential to eliminate this problem too, but its impact on the soundness of the resulting inference system and the declarative and procedural semantics of the resulting language need further investigation.

In Gödel there is no `DELAY` specification for functions. Narrowing is used to solve equations. A `DELAY` statement for functions would improve the efficiency of a computation in some cases, but would restrict the set of equations that can be solved and consequently destroy the completeness of narrowing in some other cases.

Gödel with user-defined evaluable functions is an effective workbench for narrowing strategies. Despite its lack of advanced features, such as declarative I/O and higher-order functions, it is adequate for studying in a practical setting the potential and the limitations of needed narrowing. We believe that needed narrowing will be a crucial step for the development of a general-purpose, modern, functional logic language. Some features of our proposal, their implementation techniques, and even the absence of other features may contribute to refine more ambitious proposals, such as the language Curry [9], aiming at the definition of a successor to Prolog.

Acknowledgment

We are grateful to the entire Gödel group at Bristol for designing and implementing Gödel and for assisting us in our effort to modify the compiler. We are indebted to John Lloyd for comments on and corrections of an early draft of this note.

References

[1] S. Antoy. Definitional trees. In *Proc. 3rd Conf. on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.

[3] S. Antoy, M. Hanus, R. Loogen, J. J. Moreno-Navarro, and M. Rodríguez-Artalejo. A comparison of implementations of narrowing in Prolog. In preparation.

[4] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *ESOP-86*, pages 119–132. Springer LNCS 213, 1986.

[5] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.

[6] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: a logic plus functional language. *The Journal of Computer and System Sciences*, 42:139–185, 1991.

[7] M. Hanus. Compiling logic programs with equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

[8] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[9] M. Hanus, H Kuchen, and J.J. Moreno Navarro. Curry: A truly functional logic language. In *Workshop on Laying the Foundation for a Modern Successor to Prolog*, Portland, OR, Dec. 1995. (these proceedings).

[10] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1993.

[11] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992.

[12] J. W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, University of Bristol, Dept of Computer Science, 1995.

[13] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[14] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 2nd edition, 1994.

[15] T. Suzuki, A. Middeldorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In *RTA'95*, pages 179–193, 1995. *LNCS* 914.