1000-9825/2001/12(09)1279-08 ©2001 Journal of Software 软件学报 Vol.12, No.9

Operational Semantics for Functional Logic Languages

MOHAMED Hamada

(*Language Processing Laboratory, Aizu University, Aizu-Wakamatsu City, Fukushima, Japan*)

E-mail: mohamed_hamada@hotmail.com

Received February 5, 2001; accepted April 3, 2001

Abstract: Functional languages and logic languages complement each other in the following sense. Functional programming languages, based on reduction, have properties such as deterministic evaluation and lazy evaluation; however they lack some desirable properties such as existentially quantified variables and partial data structures. On the contrary, logic programming languages, based on Horn clause logic and resolution, allow existentially quantified variables and partial data structures but lack both deterministic evaluation and lazy evaluation. From this point of view it is natural to integrate functional and logic programming languages into one paradigm. This provides a unified language with more expressive power than both logic and functional languages. This paper discusses the proposal for an operational semantics of functional logic languages, and demonstrates that the operational semantics is practically visible.

Key words: functional programming; logic programming; narrowing; unification; term rewriting; lazy narrowing

Recently there is an increasing interest in the unification of functional and logic languages. In the literature, two approaches are described to integrate functional and logic programming languages. The first approach is to integrate logic programming aspects into functional programming languages[1~3]. This approach requires allowing existentially quantified variables in expressions and using unification instead of matching in the reduction process. The other approach is to extend logic programming languages with a method to allow function definition and evaluation[4,5]. This approach requires an extension of the resolution principle which is based on syntactic unification. Both approaches to the integration have a similar operational behavior, however. The operational semantics of functional languages is rewriting and of logic languages is (syntactic) unification. From the simple fact that, narrowing = rewriting + unification, one can guess that narrowing is a natural choice as an operational semantics for functional logic languages. This observation can be emphasized by the fact that narrowing is complete for various classes of rewriting systems[6].

Narrowing was originally conceived as an E-unification (unification modulo some equational theory E) procedure in the framework of equational theorem proving[7]. Narrowing in its original definition is complex and inefficient. The reason for the complexity can be seen from the process of a narrowing step, as follows. In a single narrowing step we have to: select a subexpression to be narrowed (narex), select a rewrite rule $l\rightarrow r$ in such a way that its left-hand side l is unified with the selected narex via a most general unifier θ, and replace the narex with the instance of the right hand side r under θ. The reason for the inefficiency of narrowing is the non-determinism in narrowing steps (due to narex selection and rewrite rule selection) which results in a huge search space.

Two approaches have been pursued to improve narrowing. One approach is to introduce a restricted version of narrowing, known as narrowing strategy[8,9]. One of these strategies is basic (conditional) narrowing[10], in which narrowing steps are never applied to (sub) terms introduced by previous narrowingsubstitutions. Basic (conditional) narrowing is an important improvement over (conditional) narrowing since it results in a significant reduction of the search space. The other approaches, (e.g. Refs. [11~14]) which we pursue in our research, is to simulate narrowing by a simple set of inference rules formalized as a calculus.

In our work[15] we established an interesting connection between the two approaches. More precisely we connect (strong) completeness of our calculus with completeness of basic conditional narrowing.

We deal with conditional narrowing problems from both theoretical and practical point of view. The theoretical part, in our work, tackles the (conditional) narrowing problems (mentioned above) as follows.

To overcome the complexity problem, we decompose narrowing into more basic operations. These basic operations are represented as inference rules and the set of inference rules is formalized as a calculus. We name this calculus Lazy Conditional Narrowing Calculus) (LCNC for short). For LCNC-like calculi, soundness and completeness are important properties. Soundness means that the computed answer by the calculus is correct, while completeness means that for every solution of a given goal, a more general solution can be found by the calculus.

Soundness of LCNC is easy to show, while completeness is difficult. In our work[15~17] we established several completeness results for LCNC.

LCNC contains three sources of non-determinism: the selection of the inference rule, the selection of the equation in the goal to be solved, and the selection of the rewrite rule.

While LCNC solves the complexity problem of (conditional) narrowing and eases its implementation, the inefficiency problem remains unsolved due to the existence of nondeterminism in LCNC.

To overcome the inefficiency problem we introduced[18] a deterministic version of LCNC which we call LCNC$_d$. The practical aspect of our work demonstrates the significance of our calculus, as follows. We give a full implementation of LCNC$_d$ using Mathematica[19]. Existentially quantified equations can be solved by our calculus with respect to given rewrite rules. Thus our implementation extends the symbolic computation language Mathematica based on higher-order rewriting.

One of the immediate objectives of our implementation was to realize quick and elegant runnable calculi to study narrowing on a wide class of examples beyond hand calculated ones. However, our implementation results in a calculus that is usable in a variety of applications such as a functional logic language interpreter, in which programs are regarded as conditional term rewriting systems (CTRSs for short) and goals as a sequence of equations, as well as an equational theorem prover. In addition, our implemented calculus can be used as a research tool usable within Mathematica for studying equation solving with respect to various classes of (conditional) term rewriting systems.

In this paper we demonstrate that our deterministic lazy conditional narrowing calculus can be observed as an operational semantics for functional logic languages. We discuss an implementation of the calculus. Finally we present an application of the implemented calculus in the polymorphic type inference systems.

1 Preliminaries

In the sequel, we assume familiarity with term rewriting and narrowing, surveys can be found in Refs. [6,20, 21], however we will give some definition that we need in this paper.

First order terms $\mathcal{T}$ (terms denoted by $s, t, \ldots$, etc.) over a set of function symbols with arity (function symbols denoted by $f, g, \ldots$, etc.) and a countable set of variables $\mathcal{V}$ (variables denoted by $x, y, \ldots$, etc) is defined as the least set that satisfies the following:

- variables and constants (i.e. function symbols with arity zero) are terms, and
- if $t_1, \ldots, t_n$ are terms and f is a function symbol with arity n, then $f(t_1, \ldots, t_n)$ is a term. (f is called the head symbol of that term).

A set of variables in a term t will be denoted by $\mathcal{V}ar(t)$. An equation is a pair of terms written as $s \approx t$. A goal

G is a sequence of equations. If the sequence is empty the goal is denoted by $\square$. A substitution (denoted by θ, σ,... etc.) is a finite set of variable bindings (i.e. pairs of variables and terms) of the form $x \mapsto t$, where x is a variable and t is a term. The domain of θ is the set of its first components (i.e. variables) and the range is the set of the second components (i.e. terms). An application of a substitution θ to a term t, (denoted by $t\theta$, is the process of replacing variables of t that occurs in the domain of θ with the corresponding terms in the range of θ. This can be extended to a goal G, denoted by $G\theta$, in the obvious way.

A rewrite rule is a directed equation ($l \approx r$) denoted by $l \to r$. A conditional rewrite rule is a rewrite rule with a sequence of equations which should be solved before we apply the rule. It will be denoted by $l \to r \Leftarrow s_1 \approx t_1, \ldots, s_n \approx t_n$ (or $l \to r \Leftarrow c$ for short). For a conditional rewrite rule $l \to r \Leftarrow c$, l is the left hand side, r is the right hand side, and c is the conditional part of the rule. A (conditional) term rewrite system ((C)TRS for short), denoted by $\mathscr{R}$, is a set of (conditional) rewrite rules.

The head symbols of the left hand sides of the rewrite rules, in a term rewriting system $\mathscr{R}$, are called defined function symbols. Other function symbols are called constructors.

2 Operational Semantics

In this section we introduce our deterministic lazy conditional narrowing calculus (LCNC$_d$ for short) as an operational semantics of functional logic languages for which programs can be represented as term rewriting systems. We demonstrate that our operational semantics is practically visible by introducing a Mathematica implementation of our calculus.

2.1 LCNC$_d$

Let $\mathscr{R}$ be a CTRS. LCNC$_d$ consists of the following two groups of inference rules. The first group of LCNC$_d$ is related to the initial equations (and its descendents) and consists of the following five inference rules.

[o] outermost narrowing

$$\frac{f(s_1,\ldots,s_n)\approx t, G}{s_1 \rhd l_1,\ldots,s_n \rhd l_n, r\approx t, c, G}$$

and

$$\frac{t\approx f(s_1,\ldots,s_n), G}{s_1 \rhd l_1,\ldots,s_n \rhd l_n, r\approx t, c, G}$$

if there exists a fresh variant $f(l_1,\ldots,l_n)\to r\Leftarrow c$ of a rewrite rule in $\mathscr{R}$, where the head of t is a constructor symbol.

[i] imitation

$$\frac{f(s_1,\ldots,s_n)\approx x, G}{(s_1\approx x_1,\ldots,s_n\approx x_n, G)\theta}$$

and

$$\frac{x\approx f(s_1,\ldots,s_n), G}{(s_1\approx x_1,\ldots,s_n\approx x, G)\theta}$$

where f is a constructor symbol, $x\in \mathscr{V}ar(f(s_1,\ldots,s_n))$ or $f(s_1,\ldots,s_n)$ is not a constructor term and $\theta=\{x\mapsto f(x_1,\ldots,x_n)\}$ with $x_1,\ldots,x_n$ fresh variables.

[d] decomposition

$$\frac{f(s_1,\ldots,s_n)\approx f(t_1,\ldots,t_n), G}{s_1\approx t_1,\ldots,s_n\approx t_n, G}$$

where f is a constructor symbol,

[v] variable elimination

$$\frac{s\approx x,G}{G\theta} \quad \text{and} \quad \frac{x\approx s,G}{G\theta}$$

where s is a non-variable constructor term, $x\notin\mathcal{V}ar(s)$ and $\theta=\{x\mapsto s\}$.

[t] removal of trivial equations

$$\frac{x\approx x,G}{G}.$$

Contrary to usual narrowing, the outermost narrowing [o] generates new parameter passing equations $s_1\rhd l_1,\ldots,s_n\rhd l_n$ besides the body equation $r\approx t$ and the conditional equations c. Here we distinguish parameter passing equations (and its descendents) from the initial equations (and its descendents). We use the symbol $\rhd$ (instead of the symbol $\approx$) for the first. The second group of LCNC$_d$ is related to the parameter passing equations (and its descendents) and consists of the following three inference rules.

[o-p] outermost narrowing for parameter passing equations

$$\frac{f(s_1,\ldots,s_n)\rhd t,G}{s_1\rhd l_1,\ldots,s_n\rhd l_n,r\rhd t,c,G}$$

if there exists a fresh variant $f(l_1,\ldots,l_n)\to r\Leftarrow c$ of a rewrite rule in $\mathcal{R}$, where t is not variable,

[d-p] decomposition for parameter passing equations

$$\frac{f(s_1,\ldots,s_n)\rhd f(t_1,\ldots,t_n),G}{s_1\rhd t_1,\ldots,s_n\rhd t_n,G},$$

where f is a constructor symbol,

[v-p] variable elimination for parameter passing equations

$$\frac{s\rhd x,G}{G\theta} \quad \text{and} \quad \frac{x\rhd s,G}{G\theta}$$

where s is a non-variable term, and $\theta=\{x\mapsto s\}$.

If G and G' are the upper and lower goal in the inference rule $[\alpha]$ ($\alpha\in\{o,i,d,v,t,o\text{-}p,d\text{-}p,v\text{-}p\}$), we write $G\Rightarrow_{[\alpha]}G'$. This is called an LCNC$_d$ step. The applied rewrite rule or substitution may be supplied as subscript, that is, we write things like $G\Rightarrow_{[o],l\to r\Leftarrow c}G'$ and $G\Rightarrow_{[i],\theta}G'$. A sequence of LCNC$_d$-steps is called LCNC$_d$ derivation. A finite LCNC$_d$-derivation $G_1\Rightarrow_{\theta_1}\ldots\Rightarrow_{\theta_{n-1}}G_n$ may be abbreviated to $G_1\Rightarrow^*_\theta G_n$ where $\theta=\theta_1\ldots\theta_{n-1}$. An LCNC$_d$-refutation is an LCNC-derivation ending in the empty goal $\square$. The parameter passing equations must eventually be solved in order to obtain a solution, but we do not require that they are solved right away. That is the reason why we call the calculus lazy.

The following example shows how this calculus works.

Example. Consider the following conditional rewrite system that represents a (partial) family relationship

$$\mathbf{father}(\mathbf{brother}(x))\to\mathbf{father}(x)$$

$$\mathbf{brother}(x)\to y\Leftarrow\mathbf{father}(x)\approx z,\mathbf{father}(y)\approx z$$

$$\mathbf{grand\_father}(x)\to y\Leftarrow\mathbf{father}(x)\approx z,\ \mathbf{father}(z)\approx y$$

The following figure shows an LCNC-refutation that computes a (possible) solution $\{x\mapsto\mathbf{Jon}\}$ for the goal $\mathbf{grand\_father}(\mathbf{brother}(x))\approx\mathbf{father}(\mathbf{father}(\mathbf{Jon}))$ (the underlines represent the selected equation at each LCNC-step).

$$\mathbf{grand\_father}(\underline{\mathbf{brother}(x))\approx}\mathbf{father}(\mathbf{father}(\mathbf{Jon}))$$

$$\Downarrow_{[o]}$$

$$\underline{\mathbf{brother}(x)\rhd x_1},y\approx\mathbf{father}(\mathbf{father}(\mathbf{Jon})),\mathbf{father}(x_1)\approx z,\mathbf{father}(z)\approx y$$

$$\Downarrow_{[v\text{-}p],\{x_1\mapsto\mathbf{brother}(x)\}}$$

$$\underline{y\approx\mathbf{father}(\mathbf{father}(\mathbf{Jon}))},\mathbf{father}(\mathbf{brother}(x))\approx z,\mathbf{father}(z)\approx y$$

$$\Downarrow_{[v],\{y\mapsto\mathbf{father}(\mathbf{father}(\mathbf{Jon}))\}}$$

$$\underline{\mathbf{father}(\mathbf{brother}(x))\approx z},\mathbf{father}(z)\approx\mathbf{father}(\mathbf{father}(\mathbf{Jon}))$$

$$\Downarrow_{[v],\{z\mapsto\mathbf{father}(\mathbf{brother}(x))\}}$$

$$\begin{array}{c}
\underline{\mathbf{father}(\mathbf{father}(\mathbf{brother}(x)))\approx\mathbf{father}(\mathbf{father}(\mathbf{Jon}))}\\
\Downarrow_{[d]}\\
\underline{\mathbf{father}(\mathbf{brother}(x))}\approx\mathbf{father}(\mathbf{Jon})\\
\Downarrow_{[o]}\\
\underline{\mathbf{brother}(x)\rhd\mathbf{brother}(x_2),}\mathbf{father}(x_2)\approx\mathbf{father}(\mathbf{Jon})\\
\Downarrow_{[d\cdot p]}\\
\underline{\mathrm{x}\rhd\mathrm{x}_2,}\mathbf{father}(x_2)\approx\mathbf{father}(\mathbf{Jon})\\
\Downarrow_{[v\,p],\{x_2\mapsto x\}}\\
\underline{\mathbf{father}(x)\approx\mathbf{father}(\mathbf{Jon})}\\
\Downarrow_{[d]}\\
\underline{x\approx\mathbf{Jon}}\\
\Downarrow_{[v],\{x\mapsto \mathrm{Jon}\}}\\
\square
\end{array}$$

Although soundness of LCNC$_d$ is easy to show, its completeness is rather difficult and still under investigation for future work. However, soundness of LCNC$_d$ is sufficient in applications where we are interested in one or several solutions. For example, to reduce the complexity of finding symbolic solutions of equational constraints over arbitrary algebraic structure, a solution pattern approach was introduced in Refs. [22,23]. In this approach pattern solving is successful if some substitution of the unknown coefficients within the pattern turns the pattern into a solution.

2.2 Implementation

To avoid complex interaction between our system and Mathematica built-in conditional rewrite rules, we adopt our own implementation of conditional rewrite rules. More precisely, we defined an optimized representation of conditional rewrite rules. The idea is based on the observation that the conditional rewrite rule $f(s)\to t\Leftarrow c$ can be represented by $f(x)\to t\Leftarrow x=s,c$, where $x=s$ is a parameter passing equation, in the case of s being a non-variable term. This representation makes it possible:

1. to eliminate runtime creation of parameter passing equations, and
2. to use parameter binding mechanism of Mathematica rather than applying the variable elimination rules of our system.

The inference rules implemented directly into Mathematica rules. We have a user interface for our implementation in a form of Mathematica solvers. This solver takes as input a sequence of equations and produces as output the set of all possible solutions for these equations with respect to a given conditional term rewriting system. Existentially quantified equations can be solved by our solver with respect to given rewrite rules. Thus our system extends the symbolic computation language Mathematica based on higher-order conditional rewriting. For example, consider the one-rule term rewriting system $succ[0]\longrightarrow 1$. To solve the simple goal $succ[x]\approx 1$, which has the solution $\{x\mapsto 0\}$, by using the Mathematica solver Solve[succ[x]==1, {x}] we get the answer $\{\mathsf{x}=\mathsf{succ}^{-1}[1]\}$ which is not the desirable answer. Using our solver we get the correct answer $\{\{\mathsf{x}\to 0\}\}$, however. This shows that our implementation can also be used as an equational solver that extends Mathematica solvers.

Although LCNC$_d$ is considered as a deterministic calculus (in the sense of applying the inference rules), the selection of the rewrite rule (in case of outermost narrowing) remains a source of non-determinism. This type of non-determinism can not be avoided since it may happen that a goal has some incomparable solutions. In our system we try to simulate this non-deterministic behavior of LCNC$_d$ by applying all applicable rewrite rules.

An operation of our implemented system is achieved (by induction on goals) as follows. For each inference rule in LCNC$_d$ there is a corresponding Mathematica program. Depending on the structure of the leftmost equation in the goal, the system applies the suitable inference rule to get a new goal G. If G is empty (base case of induction) this means that a solution is found by the system and this solution will be added to the global variable AnswerList which holds all possible solutions of the given goal. If the applied inference rule was the outermost narrowing, our system will try the next applicable rewrite rule. If no applicable rewrite rules exist, the system terminates and returns the AnswerList as the set of all possible solutions of the given goal. If the system fails to find the solution(s) of the given goal, it will return the empty list as the set of solutions. In the following section we demonstrate that our system has potential to be used as an equational solver in systems which can be represented as a sequence of equations.

3 Type Checker

Polymorphic type inference systems are essential for declarative languages such as functional logic languages. In this example we describe an implementation of a slightly extended version of Hindley's type inference system (and its counterpart of Milner). We use our system's equational solver TSolve as a tool to solve equations between polymorphic types. More precisely, we generate a sequence of equations between (polymorphic) types and then TSolve is used to check consistency between these equations.

The type checker for a term is implemented as follows:

```
Tc[term_]:=Module[{teqn,tvar},{teqn,tvar}=
    TcX[term,α,{ },{α}];
    α/.TSolve[teqn,tvar][[1]]]
```

To check the type of a term t, Tc[t] is used. Tc[t] will check the type of the term t in two steps. First, it invokes the auxiliary function TcX[] (its simple implementation is ommited) with t as its first argument. Depending on the structure of t, TcX[] will form a sequence of type equations stored in teqn, and a set of type variables stored in tvar. In the second step, Tc[] invokes the term domain solver TSolve[teqn,tvar] with the arguments teqn and tvar. At this moment TSolve[teqn,tvar] will give the solution of the type equations teqn with respect to the type variables tvar. The returned result by this process will be the application of this solution to an auxiliary type variable α extracted from the term structure via TcX[] in the first step.

A (conditional) rewrite rule $l\rightarrow r\Leftarrow s_1\approx t_1,\ldots,s_n\approx t_n$ is correctly typed if all the terms $l,r,s_1,t_1,\ldots,s_n,t_n$ are typed with the same type, and all the types assigned to one variable (that appears in left-hand side, right-hand side and conditions) are one and the same. The following program can check this consistency by using TSolve. The type checker for a (conditional) rewrite rule is implemented as follows:

```
Tc[RewriteRule[s_,t_,c___]]:=
    Module[{teqn,tvar},
    {teqn,tvar}=
    TcX[XRewriteRule[s,t,{c}],α,{ },{α}];
    α/.TSolve[teqn,tvar][[1]]]
```

The operation of Tc on a (conditional) rewrite rule s→t⇐c is similar to the operation of Tc on a term t described above. The only difference is that another rule of the TcX function definition is used to extract the type equations and the type variables from the more complex structure of the rewrite rule.

4 Conclusion

In this paper we discussed our conditional narrowing calculus ($LCNC_d$) as an operational semantics of functional logic languages. We also discussed our implementation of $LCNC_d$. This work is an extension of the work of Middeldorp, *et al.*[24,25] in which they tackle narrowing problems for the unconditional term rewriting systems by introducing the lazy narrowing calculus LNC[21] and its deterministic version LNC_d[24]. In functional logic programs it is more natural to express programs as conditional term rewriting systems[26]. From a syntactic point of view the extension from unconditional to conditional case seems to be simple since we only need to add conditions wherever rewrite rules are applicable. However, this (syntactically) simple extension makes the completeness proof harder since it adds new equations (conditions) to goals in intermediate steps in the narrowing process. Conditional rewrite rules may also introduce extra variables which make the narrowing process more complex.

References:

[1] Darlington, J., Guo, Y. Narrowing and unification in functional programming-an evaluation mechanism for absolute set abstraction. LNCS 355, 1989.

[2] Silbermann, F.S.K., Jayaraman, B. Set abstraction in functional and logic programming. In: Proceedings of the 4th International Conference on Functional Programming and Computer Architecture. 1989.

[3] Silbermann, F.S.K., Jayaraman, B. A domain theoretic approach to functional and logic programming. Technical Report TUTR 91-109, Tulane University, 1991.

[4] Bonnier, S., Maluszynski, J. Towards a clean amalgamation of logic programs with external procedures. In: Proceedings of the 5th Conference on Logic Programming. MIT Press, 1988.

[5] Hanus, M. The Integration of functions into logic programming: from theory to practice. Journal of Logic Programming, 1994,19&20:583～628.

[6] Middeldorp, A., Hamoen, E. Completeness results for basic narrowing. Applicable Algebra in Engineering, Communication and Computing, 1994,5:213～253.

[7] Slagle, J.R. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. Journal of ACM, 1974,21(4):622～642.

[8] Bockmayr, A., Krischer, S., Werner, A. Narrowing strategies for arbitrary canonical systems. Fundamenta Informaticae, 1995,24(1,2):125～155.

[9] Echahed, R. On completeness of narrowing strategies. In: Proceedings of the CAAP'88. LNCS 299, 1988.

[10] Hullot, J-M. Canonical Forms and Unification. LNCS, 1980,87:318～334.

[11] Hanus, M. Efficient implementation of narrowing and rewriting. In: Proceedings International Workshop on Processing Declarative Knowledge. LNAI 567, 1991.

[12] Hölldobler, S. Foundations of Equational Logic Programming. Lecture Notes in Artificial Intelligence, 1989.

[13] Ida, T., Nakahara, K. Leftmost outside-in narrowing calculi. Journal of Functional Programming, 1997,7(2).

[14] Snyder, W. A Proof Theory for General Unification. Birkhäuser, 1991.

[15] Hamada, M., Middeldorp, A. Strong completeness of a lazy conditional narrowing calculus. Functional and Logic Programming, Singapore: World Scientific, 1997. 14～32.

[16] Hamada, M. Strong completeness of a narrowing calculus for conditional rewrite systems with extra variables. ENTCS, 2000,31(1):83～97.

[17] Hamada, M., Middeldorp, A., Suzuki, T. Completeness results for a lazy conditional narrowing calculus. Combinatorics, Computation and Logic, Springer-Verlag, 1999. 217～231.

[18] Hamada, M., Ida, T. Deterministic and non-deterministic lazy conditional narrowing and their implementations. Journal of Information Processing Society, 1998,39(3).

[19] Wolfram, S. The Mathematica Book. Wolfram Media Inc. Champaign, Illinois and Cambridge University Press 1999.

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

[21] Klop, J.W. Term rewriting systems. In: Abramsky, S., *et al.*, eds. Handbook of Logic in Computer Science. Oxford University Press, 1992. 1～116.

[22] Caprotti, O. Symbolic pattern solving for equational reasoning. In: Proceedings of the International Workshop of First-Order Theorem Proving. Austria, 1997. 53～57.

[23] Caprotti, O. Symbolic pattern solving in algebraic structures[Ph.D. Thesis]. RISC-Linz, Research Institute for Symbolic

Computation, 1997.

[24] Middeldorp, A., Okui, S. A deterministic lazy narrowing calculus. Journal of Symbolic Computation, 1998,25(6):733～757.

[25] Middeldorp, A., Okui, S., Ida, T. Lazy narrowing: strong completeness and eager variable elimination. Theoretical Computer Science, 1996,67(1,2):95～130.

[26] Hanus, M., Kuchen, H. Curry: an Integrated Functional Logic Language. (Draft version) RWTH Aachen, Germany, 1997.

函数式逻辑语言的操作语义

MOHAMED Hamada

(*Language Processing Laboratory, Aizu University, Aizu-Wakamatsu City, Fukushima, Japan*)

摘要：函数式语言和逻辑语言在下列意义上是互补的，基于归约的函数式程序设计语言具有确定和懒惰求解等性质.但同时它又缺少诸如存在量化的变量以及部分数据结构等所希望的性质.相反，基于HORN子句逻辑和消解原理的逻辑程序设计语言允许存在量化的变量和部分数据结构但又缺少确定和懒惰求解的性质.从这个角度出发，把函数和逻辑程序设计语言结合成一种范型是很自然的，这种结合提供了一种比逻辑和函数语言表达能力更强的合一语言.提出了函数式逻辑语言的操作语义，同时表明这种操作语义在实践中是可见的.

关键词：函数式程序设计；逻辑程序设计；紧缩；合一；项重写；懒惰紧缩

中图法分类号：TP311　　　**文献标识码**：A