INFORMATICA, 2000, Vol. 11, No. 1, 3–14

© 2000 *Institute of Mathematics and Informatics, Vilnius*

Theoretical Foundations of an Environment-Based Multiparadigm Language

Mario BLAŽEVIĆ, Zoran BUDIMAC, Mirjana IVANOVIĆ

Institute of Mathematics, Faculty of Science, University of Novi Sad
Trg D. Obradovića 4, 21000 Novi Sad, Yugoslavia
e-mail: bmario@eunet.yu, zjb@unsim.ns.ac.yu, mira@unsim.ns.ac.yu

Received: January 2000

Abstract. The paper presents a simple programming language and rewriting system called GENS. It is based on an extension of the λ-calculus called λ_E-calculus. GENS is a multiparadigm language: it has been used for definition of semantics and for implementation of functional, logical, procedural, and object-oriented languages. It also allows combining different programming paradigm styles in a single programming language.

The purpose of this paper is to define and to introduce the λ_E-calculus – theoretical foundation of GENS. It will also be shown how the most important langauge constructs of different programming paradigms can be defined in GENS.

Key words: λ-calculus, programming languages, rewriting systems, lambda calculus, programming paradigms, multi-paradigm.

1. Introduction and Related Work

GENS is a new programming language based on the λ_E-calculus, an extension of the λ-calculus. It is a relatively simple language, as it contains few constructs. Yet, it can easily represent semantics of programming languages belonging to different programming paradigms and their combinations.

GENS is currently implemented in programming language Oberon-2 under the Oberon operating system. It has been used for implementation of interpreters for the functional language ISWIM, for a declarative subset of Prolog (a logical language), for Pascal- (a subset of a procedural language), and for an object-oriented language Sol.

GENS was originally conceived as a generalization of the graph rewriting systems. The original "Graph reduction ENvironment System" is still a part of the GENS project, but now it's implemented on top of a simpler language that retained the old name for now, though it has little in common with graph reduction. Some other languages based on graph rewriting are Clean (Eekelen *et al.*, 1988), LEAN (Clean's predecessor (Barendregt *et al.*, 1988)), and Dactl (Glauert *et al.*, 1987; Papadopoulos, 1996).

The first version of GENS has been introduced in (Blažević and Budimac, 1997), where its compatibility with the classical graph rewriting systems has been stressed. In

(Blažević, 1998) GENS was extended with constructs similar to those presented in this paper.

A nice overview of the functional-logic multiparadigm languages and the motivations behind them can be found in (Moreno Navarro, 1995). A more ambitious language Leda, combining functional, logic and procedural paradigm, is presented in (Budd, 1995).

Other recent attempts in the field of multi-paradigm languages include combining functional languages with a kind of record-like structures. Some of them are low-level, extending λ-calculus to make it suitable for dealing with record fields. The resulting extensions have some similarities with the λ_E-calculus. Two of such extensions are the λN-calculus, presented in (Dami, 1998) and the label-selective λ-calculus, presented in (Aït-Kaci and Garrigue, 1993). The work (Dami, 1997) presents an overview of various record-calculi and compares it to the λN approach.

There have also been many attempts to use imperative input/output and other constructs in a pure functional setting. One approach, applied in functional language Haskell, can be found in (Peyton Jones and Wadler, 1993).

The next section of the paper introduces GENS. The Section 3 shortly illustrates the GENS programming. The conclusion shortly discusses GENS as the programming environment.

2. A Description of GENS

2.1. λ_E, *the Calculus Underlying GENS*

The λ_E-calculus described here is an extension of λ-calculus. Beside the names, applications and λ-abstractions, inherited from λ-calculus, this new calculus includes *environment* as a new kind of term.

As usual in programming, environment is a set of pairs (name, value), also called *attributes*, and represents a mapping from the set of names (labels) to the set of their possible values (all terms). This mapping can be partial. If the environment σ contains the pair (l, t) we say that σ *defines* label l, and that it *assigns* value t to l.

EXAMPLE 1. An example of environment is $\{(a, b), (b, (a\ b)), (c, \{(a, c)\})\}$. This environment defines three labels: a, b and c. It assigns to label a another label b, to b the application $a\ b$, and to c another environment {(a, c)}.

The mathematical set notation is not very appropriate for environments, so this notation is shortened in GENS. Pairs are written as $l = t$ instead of (l, t) and the $\{\}$ brackets are replaced with $()$. So this environment would be written as $(a = b, b = (a\ b), c = (a = c))$.

If two environments σ_1 and σ_2 are given, we can define the operation of asymmetric union upon them, written as $\sigma_1 \triangleright \sigma_2$. This operation is similar to set union, but it preserves the uniqueness of the attribute labels making the result a new partial mapping. The first

environment σ_1 has a greater priority, so all the attributes from the second environment σ_2 in collision with the first get discarded from result:

$$\sigma_1 \triangleright \sigma_2 = \sigma_1 \cup (\sigma_2|_{D(\sigma_2)\setminus D(\sigma_1)}).$$

$D(\sigma)$ denotes the domain set of environment σ, which is the set of all names σ defines.

EXAMPLE 2. If we are given the environments $\sigma_1 = (a{=}\,1, b{=}\,2)$ and $\sigma_2 = (b{=}\,3, c{=}\,4)$ then their asymmetric union is $\sigma_1 \triangleright \sigma_2 = (a = 1, b = 2, c = 4)$. In this case there was no collision for the attributes $a = 1$ and $c = 4$, while the attribute $b = 3$ was taken from σ_1.

The set of λ_E-calculus terms consists of names, applications, abstractions, and environments. The set of names is split into labels and variables, L and V. Only labels can be defined by any environment. Labels are free names, and variables are always bound in some enclosing lambda-abstraction. Application is associative: the terms $(t_1\ t_2)t_3$ and $t_1(t_2\ t_3)$ are equivalent.

$$T = L \cup V \cup \{t_1\ t_2 |\ t_1, t_2 \in T\} \cup \{\sigma |\ \sigma : L \mapsto T\} \cup$$
$$\cup \{\lambda l/v.t |\ l \in L, v \in V, t \in T\} \cup \{\lambda */v.t |\ v \in V, t \in T\}.$$

The λ_E-calculus reduction rules take place only when a term gets applied to an environment, which means that the redex must be in form $t\ \sigma$. Reduction depends on the type of the first term t. In the rules presented below l always denotes a label, v a variable, σ, σ_1, and σ_2 are environments, and t is any term.

$$\sigma_1\ \sigma_2 \to \sigma_1 \triangleright \sigma_2, \tag{1}$$
$$l\ \sigma \to \sigma(l)\ \sigma, \qquad \text{if } l \in D(\sigma), \tag{2}$$
$$\lambda\, l/v.t\ \sigma \to t[\sigma(l)/v]\ \sigma, \qquad \text{if } l \in D(\sigma), \tag{3}$$
$$\lambda */v.t\ \sigma \to t[\sigma/v]\ \sigma, \qquad \text{if } D(\sigma) = L. \tag{4}$$

The demand $D(\sigma) = L$ in (4) could be relaxed, at the price of either complicating the semantics or destroying its confluence. It should also be noted that the rewriting rule (2) is not really necessary, because we can always replace an application $l\ \sigma$ with the similar term $\lambda l/v.v\ \sigma$ that also reduces to $\sigma(l)\ \sigma$. However, the former application seems to be very convenient and natural, and it appears so often that it deserves its special treatment.

The meaning of the replacement $t[t'/v]$ is similar to its meaning in λ-calculus. Only the variables ever get renamed by the α-conversion, the labels are immutable:

$$\begin{aligned} &l[t/v] = l, \\ &v[t/v] = t, \\ &v'[t/v] = v', \qquad\qquad \text{if } v \neq v', \end{aligned}$$

$$\begin{aligned} &(t_1\ t_2)[t/v] = t_1[t/v]\ t_2[t/v], \\ &\sigma[t/v] = \{(l, t'[t/v]) |\ (l, t') \in \sigma\}, \end{aligned}$$

$\lambda l/v.t'\ [t/v] = \lambda l/v.t',$
$\lambda l/v'.t'\ [t/v] = \lambda l/v'.\,(t'\ [t/v])\,,$ where $v \neq v'$ and $v' \notin FV(t),$
$\lambda*/v.t'\ [t/v] = \lambda*/v.t',$
$\lambda*/v'.t'\ [t/v] = \lambda*/v'.\,(t'\ [t/v])\,,$ where $v \neq v'$ and $v' \notin FV(t).$

Theorem 1. *The λ_E-calculus has the Church-Rosser property.*

The proof is similar to the classical proof for the λ-calculus, but considerably longer because of four different reduction rules compared to one rule in λ-calculus.

Now we can show some examples of reduction. The symbol $\rightarrow^*$ denotes the reduction to the normal form:

EXAMPLE 3.

$$\begin{array}{ll}
a(b{=}\ c) & \rightarrow^*\ a(b{=}\ c),\\
a(a{=}\ c) & \rightarrow^*\ c(a{=}\ c),\\
\lambda a/v.v\ (b{=}\ c) & \rightarrow^*\ \lambda a/v.v\ (b{=}\ c),\\
\lambda a/v.v\ (a{=}\ c) & \rightarrow^*\ c(a{=}\ c),\\
(a{=}\ 1, b{=}\ 2)\ (b{=}\ 3, c{=}\ 4) & \rightarrow^*\ (a{=}\ 1, b{=}\ 2, c{=}\ 4),\\
a(a{=}\ b, b{=}\ c) & \rightarrow^*\ c(a{=}\ b, b{=}\ c),\\
a(a{=}\ b(b{=}\ c)) & \rightarrow^*\ c(a{=}\ b(b{=}\ c), b{=}\ c),\\
\lambda a/v.(b(d{=}\ v))\ (a{=}\ c) & \rightarrow^*\ b(a{=}\ c, d{=}\ c).
\end{array}$$

2.2. *Initial Environment*

The environment can be seen as a library of named functions. This feature makes it possible to define recursive functions without the Y-combinator trick. Besides, extending λ_E-calculus with predefined functions is quite easy. If we need to use some functions in a term t, we simply apply this term to the environment σ which defines these functions and then we reduce the application $t\ \sigma$ instead of t itself.

The language GENS is just the λ_E-calculus where all terms are applied to a special initial environment we shall call σ_{Gens}. Its definition follows.

$$\begin{array}{l}
\sigma_{Gens}\ = (\\
\quad Cont{=}\ (),\\
\quad Let{=}\ Cont,\\
\quad LastLabel{=}\ Let,\\
\quad Fail{=}\ (LastLabel{=}\ Fail),\\
\quad Seq{=}\ \lambda Cont/c.\,\lambda Right/r.\,Left(Cont{=}\ r(Cont{=}\ c)),\\
\quad Field{=}\ \lambda*/env0.\,\lambda Right/r.\,Seq(Right{=}\ \lambda*/env1.\\
\qquad Seq(Left{=}\ r, Right{=}\ \lambda*/env2.\\
\qquad\quad Fold(E0{=}\ env0, E1{=}\ env1, E2{=}\ env2))),
\end{array}$$

$$Fold = [Fold],$$
$$Dis = \lambda */env.\ \lambda Right/r.Left(Fail = r\ env)$$
$$) \triangleright \bigcup_{l \in L} (l = Cont\,(LastLabel = l)).$$

The σ_{Gens} environment defines all the labels in L. Most of them are assigned the default value of $Cont(LastLabel = l)$, which means they reduce to the label $Cont$ after assigning itself to the label $LastLabel$. All these labels with default value from now on will be called "undefined", since there are no undefined labels by the old meaning of the word.

The σ_{Gens} environment also defines several more labels that have a more important role. There are many other useful functions introduced in σ_{Gens}, but the ones shown are essential for extension of the calculus to a practical programming language:

- Label $Cont$ is assigned the current continuation.
- Seq contains the sequencing function, expecting its two arguments in labels $Left$ and $Right$. This function reduces to the left term (the one assigned to the label $Left$), putting the right term on top of the continuation. The effect is that the right term gets applied to the environment resulting from the reduction of the left term. In other words, Seq could also be defined simply as $Seq = \lambda Left/l.\ \lambda Right/r.\ (r\ l)$.
- $Field$ is similar to Seq, except that the environment resulting from the left term is discarded from the final result. To produce this result, this function calls the primitive function $Fold$. $Fold$ expects three environments σ_0, σ_1 and σ_2 in labels $E0$, $E1$ and $E2$, respectively, and reduces to $\sigma_0|_{D(\sigma_1 \cap \sigma_2)}$.
- The function assigned to Dis represents disjunction of its two arguments expected in labels $Left$ and $Right$. It reduces to the left argument, assigning the right one to the label $Fail$.
- Label $Fail$ is the failure label. If any disjunctions have been reduced, it contains the alternatives that are to be tried if the current reduction fails.
- $LastLabel$ contains the last reduced undefined label.
- Let behaves as an undefined label, except that it doesn't change the value of $LastLabel$.

2.3. *Syntax Extensions*

The language GENS is based on λ_E-calculus extended with the initial environment σ_{Gens} defined above and some syntax sugar to ease its use. The EBNF syntax of GENS, abstracted from some details such as primitive data type and priority, is presented below:

```
Term        = Name | Application | Abstraction |
            Sequence | Field | Disjunction.
Application = Term Environment.
```

```
Environment = '(' Attributes] ')'.
Attributes  = Attribute {',' Attribute}.
Attribute   = [Name '='] Term.
Abstraction = '\' Abstracted {',' Abstracted} '->' Term.
Abstracted  = Name ['/' Name].

Sequence    = Term ';' Term.
Field       = Term '.' Term.
Disjunction = Term '|' Term.
```

Though it is larger, GENS syntax is actually more restrictive than the λ_E syntax. The changes from the λ_E syntax are the following:

- The environment attributes can now be written in a shorter form, without the left-hand side "$Label=$". In that case their label is assumed to be equal to the ordinal position of the attribute in the environment. The first attribute has default label $1st$, the second $2nd$ etc. For example, the environment $(3, b, a = 9)$ is a shorter form for $(1st = 3, 2nd = b, a = 9)$.
- An environment can appear only at the right-hand side of an application. This restriction was necessary to ensure that all valid GENS terms reduce the continuation. Instead of an environment σ we can use the application $Let\ \sigma$, which behaves much the same but also reduces the continuation.
- There is no special syntax to distinguish between labels and variables. If a name is free, then it's considered a label. If a name appears in the place for variable in an abstraction ($\lambda l/\underline{v}.t$), then it is a bound variable and all its occurrences in the scope of this abstraction (except on the attribute left-hand side) are variables, not labels.
- Abstraction is written with '$\backslash$' instead of λ and '$->$' instead of dot, as usual in functional languages. Several comma-separated names can be abstracted. If only a single name is written instead of a pair $name_1/name_2$, then it's assumed that both the label and the variable are written the same. And finally, there is no equivalent to $\lambda *//v.t$. Only a single label can be abstracted.
- Three new constructs are added:
  - Sequence $\langle Term_1 \rangle ; \langle Term_2 \rangle$ is a shorthand for $Seq(Left = \langle Term_1 \rangle , Right = \langle Term_2 \rangle)$.
  - The field construct $\langle Term_1 \rangle . \langle Term_2 \rangle$ is a shorthand for $Field(Left = \langle Term_1 \rangle , Right = \langle Term_2 \rangle)$.
  - Disjunction $\langle Term_1 \rangle \mid \langle Term_2 \rangle$ is a shorthand for $Dis(Left = \langle Term_1 \rangle , Right = \langle Term_2 \rangle)$.

The result of the term reduction is always a large environment. To make the output simpler, only the difference between the resulting environment and σ_{Gens} get printed. The values of the predefined system labels are also excluded from it, and the label assigned to $LastLabel$ is taken in front:

$$Output[\sigma] = \sigma(LastLabel)\ (\sigma_{L \setminus \{Cont, Fail, Left, Right, LastLabel\}} \setminus \sigma_{Gens}).$$

In this way the reduction is taken "one step back", because if we apply the output to σ_{Gens} we get the same result again.

EXAMPLE 4.

$$Output\left[\left(a=5, b=c, LastLabel=c, Cont=()\right)\right] = c(a=5, b=c).$$

2.4. *The Use of Constructs*

The sequence construct is well known from imperative languages. The rewriting of the sequence $Term_1; Term_2$ is done by first reducing $Term_1$ to its root normal form σ_1'. If no failure occurred, then we proceed with rewriting of the application $Term_2\ \sigma_1'$. Multiple sequence $Term_1; Term_2; \ldots; Term_N$ is grouped to the right, as $Term_1; (Term_2; \ldots; Term_N)$. Each subsequent term adds to the environment passed from left to the right.

Field construct behaves in a similar way. The first term is also reduced to the root normal form and the resulting environment is passed to the right term before its rewriting. Contrary to the sequence construct, the passed environment gets discarded from the final rewriting result. Field can be thought of as analogy of the record field access or the method call from the object-oriented languages.

The behaviour of these two constructs can be seen from examples. The relation $\xrightarrow[Gens]{}{}^*$ shows the actual output of the GENS interpreter on the right hand side for the input given on the left hand side. It can be defined as

$$t_{in} \xrightarrow[Gens]{}{}^* t_{out} \overset{def}{\iff} t_{in}\ \sigma_{Gens} \rightarrow^* \sigma\ \wedge\ Output[\sigma] = t_{out}.$$

EXAMPLE 5.

$$\begin{aligned}
&a; b && \xrightarrow[Gens]{}{}^*\ b(),\\
&a(c=d); b && \xrightarrow[Gens]{}{}^*\ b(c=d),\\
&a(c=d); c && \xrightarrow[Gens]{}{}^*\ d(c=d),\\
&a(c=d); b(c=e) && \xrightarrow[Gens]{}{}^*\ b(c=e),\\
&a(c=d); b(e=c) && \xrightarrow[Gens]{}{}^*\ b(c=d, e=c),\\
&a(c=d); \backslash c- > b(e=c) && \xrightarrow[Gens]{}{}^*\ b(c=d, e=d).
\end{aligned}$$

EXAMPLE 6.

$$
\begin{array}{ll}
a.b & \xrightarrow[Gens]{}{}^{*}\ b(), \\
a(c{=}\,d).b & \xrightarrow[Gens]{}{}^{*}\ b(), \\
a(c{=}\,d).c & \xrightarrow[Gens]{}{}^{*}\ d(), \\
a(c{=}\,d).b(c{=}\,e) & \xrightarrow[Gens]{}{}^{*}\ b(c{=}\,e), \\
a(c{=}\,d).b(e{=}\,c) & \xrightarrow[Gens]{}{}^{*}\ b(e{=}\,c), \\
a(c{=}\,d).\backslash c{-}{>}\, b(e{=}\,c) & \xrightarrow[Gens]{}{}^{*}\ b(e{=}\,d).
\end{array}
$$

The elements of GENS described so far suffice to describe not only a simple command sequence from imperative languages, but also the branching, loops and other control structures. However, the behaviour of exception is very hard to describe as an instruction sequence – it is practically impossible to add exceptions to an imperative language without substantial modification of its semantics.

To raise an exception in GENS, we call the label $Fail$ that has a special treatment by the control constructs. When a term invokes $Fail$, we say that the rewriting failed. Now, if the rewriting of the first term in the sequence $Term_1; Term_2$ fails, the rewriting is not continued to $Term_2$ but stopped. The rewriting of sequence results with $Fail$. The same holds for field.

There still remains the problem of "catching" and correcting the exception, which is the role of disjunction $Term_1 \mid Term_2$.

If $Term_1$, together with the continuation of disjunction, reduces to anything other than $Fail$, that is the result of the whole disjunction. If rewriting of $Term_1$ fails, then the normal form of disjunction is equal to the normal form of $Term_2$. Disjunction is used to correct the local failure and to finally reduce the whole expression successfully.

EXAMPLE 7.

$$
\begin{array}{ll}
Fail; a & \xrightarrow[Gens]{}{}^{*}\ Fail(), \\
Fail.a & \xrightarrow[Gens]{}{}^{*}\ Fail(), \\
a \mid b & \xrightarrow[Gens]{}{}^{*}\ a(), \\
Fail \mid b & \xrightarrow[Gens]{}{}^{*}\ b(), \\
(a(b{=}\,Fail) \mid a(b{=}\,c)); b & \xrightarrow[Gens]{}{}^{*}\ c(b{=}\,c).
\end{array}
$$

The last example shows that a disjunction remains effective even if a failure happens outside of its scope. This feature is similar to backtracking in Prolog.

2.5. *Primitive Operations and Data Types*

Every programming language has some built-in basic data types. In GENS they include 32-bit integers, characters, strings and texts (textual files). This choice proved sufficient for the basic purpose of the language – playing with the programming languages.

Primitive data types can not be rewritten, which means they are always in normal form. Contrary to labels, they can't be assigned a value by any environment. The only way to handle them is through the built-in functions.

Every built-in operation is assigned to one label in the initial environment. There are also some reserved labels that are not assigned any special value. They are used instead as the argument-holders for other built-in functions. These are for example $Value$, $Property$, $1st$, etc. The argument names of the predefined operations are fixed.

All built-in operations of GENS can be roughly divided in three groups: low-level operations, operations on basic data types and parser operations. We shall shortly describe only the most intresting functions that will be used in the rest of the paper.

Low-level operations are used for elementary changes of the environment:

- Seq, $Field$ and Dis are the root labels of sequence, field and disjunction. They have been already discussed.
- $Reduce$ reduces the value of the name $Value$ to its root normal form using the current environment. It can be defined as:

$$Reduce\,\sigma \rightarrow (Value{=}\,t'')\,\sigma, \quad \text{where } t'' = Output[t'] \ \wedge\ Value\,\sigma \rightarrow^{*} t'.$$

- Set assigns the value of label $Value$ to the label assigned to label $Property$:

$$Set(Property{=}\,p, Value{=}\,v) \rightarrow (p{=}\,v).$$

- $FreshLabel$ creates a new label and assigns it to name $Value$.
- $System$ holds the initial system environment σ_{Gens}, which defines all the operations given here.

The following functions work with primitive data types:

- Add, Sub, Mul, and Div are the functions for adding, subtracting, multiplying and dividing integers. The expected parameters are the values of names $1st$ and $2nd$. These values are reduced to root normal form before the operation, which means the functions are strict.
- $Equal$, $Less$, $Greater$, $LessEq$, $GrEq$, and $Different$ are predicates which compare the value of name $1st$ with the value of name $2nd$. In case the relation is satisfied the result is label $True$, and otherwise the rewriting fails. These predicates can all be applied to character, numerical, and string types, and predicates $Equal$ and $Different$ to all terms.

3. A Short Example

Using a short example it will be shown how GENS can be used in programming. Other examples are out of the intended scope of this paper and will be published elsewhere.

The let-construct known from functional programming languages can be simulated with the already mentioned label Let. It can be used in a kind of an imperative programming style:

$$\begin{array}{l} Let(\\ \quad a = 3, \\ \quad b = 5); \\ Add(a, b) \end{array} \quad \xrightarrow[Gens]{}{}^{*} 8$$

$$\begin{array}{l} Let(\\ \quad a = 3, \\ \quad b = 5); \\ \backslash a, b - > Let(\\ \quad c = Add(a, b)) \end{array} \quad \xrightarrow[Gens]{}{}^{*} \quad \begin{array}{l} Let(\\ \quad a = 3, \\ \quad b = 5, \\ \quad c = Add(3, 5)) \end{array}$$

4. Conclusion

To ease the use of GENS, a specialized language G has been developed. G is a language similar in form to BNF. G is written in GENS and its purpose is to define syntax of a programming language and the translation of a program to its syntax tree, which is an equivalent GENS program.

Using G, several programming languages have been implemented as interpreters: ISWIM, Prolog, Pascal-, and Sol. Beside them, the lambda calculus and the "classical" graph rewriting system have been implemented as well. All implemented languages can be used for specification of other languages' semantics thus forming generations of languages emerging from GENS.

The translation of a programming language (say P) is fully automated. When a P program is demanded, it is loaded into the system through the P syntax definition written in G. Then its translation down the chain of languages is automatically invoked. Once created, parser for a language P is then preserved so that every following translation is done directly.

In this way GENS was extended to an integrated programming environment for programming language development. Further research will be concentrated onto support for persistence, compiling, and other issues necessary for development of practical programming languages.

References

Aït-Kaci, H., J. Garrigue (1993). Label-selective lambda-calculus. In *Proceedings of the 13th International Conference on Foundations of Software Technology and Theoretical Computer Science*, Bombay.

Barendregt, H., M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer, M. Sleep (1988). LEAN – an intermediate language based on graph rewriting. *Parallel Computing*, **9**, 163–177.

Blažević, M., Z. Budimac (1997). Attributed graph rewriting system. In *Proc. of XII Conference on Applied Mathematics*, Subotica, Yugoslavia, in print.

Blažević, M. (1998). Reduction of attributed graphs. In *Proc. of ETRAN Conference*, Zlatibor, Yugoslavia, in print (in Serbian).

Blažević, M. (1998). *Implementing Prolog using Attributed Graph Reduction*. Seminar paper, Inst. of Mathematics, Faculty of Science, Univ. of Novi Sad (in Serbian).

Budd, T. A. (1995). *Multiparadigm Programming in Leda*. Addison-Wesley.

Dami, L. (1998). A lambda-calculus for dynamic binding. *Theoretical Computer Science*, Special Issue on Coordination, Feb.

Dami, L. (1997). A comparison of record- and name-calculi. In D. Tsichritzis (Ed.), *Objects at Large*, Centre Universitaire d'Informatique, University of Geneva, July.

Eekelen, M. van, H. Huitema, E. Nöcker, M. Plasmeijer, J. Smetsers (1988). Concurrent clean – an intermediate language based on graph rewriting, *Parallel Computing*, **9**, 163–177.

Glauert, J., J. Kennaway, M. Sleep (1987). DACTL: a computational model and compiler target language based on graph reduction. *ICL Technical J.*, **5**, 509–537.

Moreno Navarro, J., J. (1995). Expressivity of functional – logic languages and their implementation

Papadopoulos, G. (1996). *Concurrent Object-Oriented Programming Using Term Graph Rewriting Techniques*. Information and Software Technology.

Peyton Jones, S., P. Wadler (1993). Imperative functional programming. In *Proc. of 20th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January.

M. Blažević received MSc degree in Computer Science from Novi Sad University in 1999. Now he is PhD student at the same university. His scientific interests include, programming languages, especially functional programming languages, multi-paradigm environments, databases and information systems.

Z. Budimac received MSc degree in Computer Science from Novi Sad University in 1991 and PhD degree in Computer Science from the same university in 1994. Presently he is associate professor at Institute of Mathematics, Faculty of Science, University of Novi Sad. His scientific interests include, programming languages, especially agent oriented and distributed programming, mobile technology, software engineering, functional programming languages, operating systems.

M. Ivanović received MSc degree in Computer Science from Novi Sad University in 1988 and PhD degree in Computer Science from the same university in 1992. Presently she is associate professor at Institute of Mathematics, Faculty of Science University of Novi Sad. Her scientific interests include, programming languages, especially agent oriented programming with application in workflow and education processes, software engineering, object oriented systems, compilers.

Teoriniai aplinkos savybėmis grindžiamos daugiaparadigmės kalbos pagrindai

Mario BLAŽEVIĆ, Zoran BUDIMAC, Mirjana IVANOVIĆ

Straipsnyje aprašyta paprasta perrašymo sistemos tipo programavimo kalba GENS. Teorinis kalbos pagrindas yra specialius λ skaičiuotės plėtinys λ_E. Pagrindinė straipsnio paskirtis – apibrėžti šį plėtinį ir parodyti, kaip skirtingų paradigmų kalbų konstrukcijas aprašyti GENS kalba. GENS tinkama funkcinių, loginių, procedūrinių ir objektinių kalbų bei kalbų, sudarytų derinant šias paradigmas, semantikai aprašyti. Ji gali būti vartojama ir kaip išvardinto tipo kalbų įgyvendinimo kalba.