NOVI SAD J. MATH.
VOL. 29, NO. 2 (1999), 13–27
PROC. VIII INT. CONF.
"ALGEBRA & LOGIC" (NOVI SAD, 1998)

FORMAL SPECIFICATION OF A GRAPH REWRITING SYSTEM WITH ENVIRONMENTS

Mario Blažević, Zoran Budimac and Mirjana Ivanović
Institute of Mathematics, University of Novi Sad
Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia
e-mail: *bmario@eunet.yu, {zjb,mira}@unsim.ns.ac.yu*

Abstract

The paper presents a programming language and rewriting system called GENS (short for Graph ENvironment System). GENS is an extension of graph rewriting systems, with addition of new concepts (attributes and environments). GENS can be used for definition of programming language semantics and for implementation of programming languages. It is also possible to mix features from different programming paradigms into a single programming language. GENS has been used for experimental implementation of several programming languages.

AMS Math. Subject Classification: 68N15, 68Q42
Key words and phrases: programming languages, rewriting systems

1. Introduction and related work

GENS (short for Graph ENvironment System) is a programming language described and implemented as a graph rewriting system. It is simple and contains only several language contructs. Yet, it can represent semantics of programming languages of several programming paradigms and their combinations. GENS can also be used for implementation of these languages.

GENS has been implemented in programming language Oberon-2 under Oberon operating system. It has been used for implementation of several programming languages: ISWIM (functional language), subset of Prolog (logical language), Pascal (subset of a procedural language) and their various combinations. The implementation of a simple object-oriented language is currently underway.

GENS has originated from graph rewriting systems. Two best known languages utilizing graph rewriting are Clean [9], LEAN (Clean's predecessor [2]), and DACTL [10, 13].

GENS has been introduced in [3], where its compatibility with classical graph rewriting systems has been stressed. In [4] GENS was extended with constructs similar to those presented in this paper.

Other recent attempts in the field of multi-paradigm languages include combining functional languages with record-like structures. Some of them are low-level, extending λ-calculus to make it suitable for dealing with record fields. Two such approaches have been presented in [7] and [1]. The work [8] presents an overview of various record-calculi and compares it with an extended-λ approach.

Another field of research is dealing with contexts and explicit substitutions as opposed to β-substitutions of the λ-calculus. Explicit substitution is the way the lambda abstraction works in GENS. There is a presentation in [11] of a calculus combining the explicit and β-substitutions.

A nice overview of the functional-logic languages and the motivations behind them can be found in [12]. A more ambitious language Leda, combining functional, logic and imperative paradigm, is presented in [6].

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 [14].

2. A description of GENS

Though simple, the language is a bit unusual. Therefore it will be described in several steps, from the basic language constructs to more advanced features. In each step both syntax and semantics of a construct will be given. The syntax will be defined in Backus-Naur Form (BNF), while the semantics will be defined by rewriting rules. We also introduce the following notation: $\rightarrow$ denotes one rewriting step, while the $\rightarrow^*$ denotes the rewriting to the

root normal form.

This section describes only the base language with no syntax extensions, while the last section shortly explains how GENS can be extended with new syntax constructs if needed.

2.1 The basic language element: label

The basic building element of GENS is a label (name), written down as a string of letters, digits, and underscores.

```
Label =   (Letter | Digit | '_') Label.
```

A stand-alone label is called a variable. A label can not be reduced on its own, out of some environment which gives it a value.

2.2 Attributes and environment

```
Term          =   Label | Environment.
Environment   =   Label '(' [Attributes] ')'.
Attributes    =   Attribute [',' Attributes].
Attribute     =   Label '=' Term.
```

As the syntax above shows, environment is a term of the form:

$$Root(Label_1 = Value_1, Label_2 = Value_2, \ldots)$$

where the *Root* and $Label_i$ are labels, and $Value_i$, $i = 1, 2, \ldots$ are terms. Formally, we have

Definition 1. *Attribute is an ordered pair (L, V) where L is drawn from the set of labels and V from the set of terms. It is written as $L = V$. The L is called the attribute label, and the term V is its value.*

Definition 2. *Environment ρ is an ordered pair (t_ρ, σ_ρ) consisting of a term t_ρ and an attribute set σ_ρ. For the purpose of the semantics definitions we shall denote it as $\langle t_\rho / \sigma_\rho \rangle$, while in the GENS syntax we write it in a more appropriate way as $t_\rho(\sigma_\rho)$. The term t_ρ is the root of the environment ρ.*

According to the GENS syntax only label can appear as the root of the environment. However, during its rewriting root can be an arbitrary term.

Labels of the attributes in the set σ_ρ must be all different. In mathematical terms, one environment's attributes make a partial mapping from the set of labels to the set of all terms. If for a label L holds that $L \in D_\rho$, where D_ρ is the domain of the mapping σ_ρ, we say that the label L is defined by the environment ρ, or that ρ defines L. In that case we define the local value of the label L in the environment ρ as $\sigma_\rho(L)$. The mapping is written as the comma-separated list of its attributes.

If two environment mappings σ_1 and σ_2 are given, we can define the operation of the asymmetric union or attribute inheritance 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 so the result remains to be a partial mapping. The first mapping σ_1 has a greater priority, so all the attributes from the second mapping σ_2 in collision with the first get discarded from the result:

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

Example 1. *If the mappings σ_1, σ_2 are given by $\sigma_1 = \{(a,1),(b,2)\}$ and $\sigma_2 = \{(b,3),(c,4)\}$, then $\sigma_1 \triangleright \sigma_2 = \{(a,1),(b,2),(c,4)\}$. In this case there was no collision for the attributes labeled a and c while the attribute labeled b was taken from σ_1.*

We also need to introduce a kind of delayed inheritance, which is denoted as $\sigma_1 : \sigma_2$. This is not an operation on two mappings, but a constructor of a new data type which behaves exactly the same as $\sigma_1 \triangleright \sigma_2$ but preserves its own structure:

$$\begin{array}{lcl} D(\sigma_1 : \sigma_2) & = & D(\sigma_1) \cup D(\sigma_2) \\ (\sigma_1 : \sigma_2)(l) & = & (\sigma_1 \triangleright \sigma_2)(l) \\ \sigma_1 \triangleright (\sigma_2 : \sigma_3) & = & (\sigma_1 \triangleright \sigma_2) : \sigma_3 \end{array}$$

Example 2. *To illustrate the rules for environment rewriting, we show first some examples.*

$$\begin{array}{lcl} a & \to^* & a \\ a() & \to^* & a() \\ a(b=c) & \to^* & a(b=c) \\ a(a=c) & \to^* & c(a=c) \\ a(b=c,d=e) & \to^* & a(b=c,d=e) \\ a(b=c,a=e) & \to^* & e(b=c,a=e) \\ a(a=b,b=c) & \to^* & c(a=b,b=c) \\ a(a=b(b=c)) & \to^* & c(a=b(b=c),b=c) \end{array}$$

An environment can not be rewritten unless it defines its root. If, on the contrary, the local value of the root in this environment is V (then the environment is of the form: $R(\ldots, R = V, \ldots)$ where R is the environment root), it can be reduced to the term $V(\ldots, R = V, \ldots)$. The new root V could be an environment itself, and in that case next rewriting step should unify the attributes in two mappings. If we write them more formally, the rewriting rules are very short:

$$T = L \cup \{\langle t/\sigma \rangle \,|\, t \in T, \sigma : L \mapsto T\}$$

$$\begin{array}{lcl} \langle l/\sigma \rangle & \rightarrow & \langle \sigma(l)/\sigma \rangle \text{ , if } l \in D(\sigma) \\ \langle \langle t/\sigma \rangle / \sigma' \rangle & \rightarrow & \langle t/\sigma \triangleright \sigma' \rangle \end{array}$$

where l and L represent a label and the set of all labels, while t and T represent any term and the set of all terms, respectively. The same notation holds for other semantics definitions.

2.3 Lambda abstraction

```
Term          =   Label | Environment | Abstraction.
Abstraction   =   '\' LocalNames  '->' Term.
LocalNames    =   Label [',' LocalNames ].
```

Lambda abstraction is a well-known language construct from the classic λ-calculus and from functional languages. Even its syntax in GENS is exactly the same as in functional languages. But its semantics is different because GENS does not have the application construct which the λ-calculus is based on

Example 3.

$$\begin{array}{ll}
\backslash a- > b & \rightarrow^* \ \backslash a- > b \\
\backslash a- > a & \rightarrow^* \ \backslash a- > a \\
c(c = \backslash a- > a, d = e) & \rightarrow^* \ (\backslash a- > a)(c = \backslash a- > a, d = e) \\
c(c = \backslash a- > a, a = e) & \rightarrow^* \ e(c = \backslash a- > a, a = e) \\
c(c = \backslash a- > a(), a = e) & \rightarrow^* \ (\backslash a- > a())(c = \backslash a- > a(), a = e) \\
c(c = \backslash a- > b(d = a), a = e) & \rightarrow^* \ b(c = \backslash a- > b(d = a), a = e, d = e) \\
c(c = \backslash a, b- > e(f = a), & \rightarrow^* \ e(a = b, b = d, f = b, \\
a = b, b = d) & \qquad c = \backslash a, b- > e(f = a)) \\
c(c = \backslash a- > \backslash b- > e(f = a), & \rightarrow^* \ e(a = b, b = d, f = d, \\
a = b, b = d) & \qquad c = \backslash a, b- > e(f = a))
\end{array}$$

A lambda abstraction can not be rewritten on its own. It is rewritten only in the root of some environment which defines all its local names. In that case the rewriting is done by replacing all the appearances of the local names in the body of the lambda abstraction by their values in the given environment. In a more formal notation, the rewriting looks like this:

$$T = L \cup \{\langle t/\sigma\rangle \,|t \in T, \sigma : L \mapsto T\} \cup \{\lambda l_1, l_2, \ldots, l_n- > t | l \in L, t \in T\},$$

$$\begin{array}{lcl} \langle l/\sigma\rangle & \rightarrow & \langle \sigma(l)/\sigma\rangle \text{ , if } l \in D(\sigma) \\ \langle\langle t/\sigma\rangle/\sigma'\rangle & \rightarrow & \langle t/\sigma \triangleright \sigma'\rangle \\ \langle \lambda l_1, l_2, \ldots, l_n- > t/\sigma\rangle & \rightarrow & < t[l_1 = \sigma(l_1), l_2 = \sigma(l_2), \ldots \\ & & \ldots, l_n = \sigma(l_n)]/\sigma > \text{ , if } l_1, \ldots, l_n \in D(\sigma) \end{array}$$

The meaning of the replacement $t\,[l_1 = t_1, l_2 = t_2, \ldots, l_n = t_n]$, or shorter $t\,[\sigma]$, is similar to its meaning in λ-calculus and functional languages:

$$\begin{array}{lcl} l[\sigma] & = & l, \text{ if } l \notin D(\sigma) \\ l[\sigma] & = & \sigma(l), \text{ if } l \in D(\sigma) \\ \langle t/\sigma\rangle\,[\sigma'] & = & \langle t/\sigma[\sigma']\rangle \\ \sigma[\sigma'] & = & \{(l, t[\sigma'])|(l,t) \in \sigma\} \\ (\lambda l_1, l_2, \ldots, l_n- > t)[\sigma] & = & \lambda l_1, l_2, \ldots, l_n- > (t[\sigma|_{D(\sigma)\setminus\{l_1,l_2,\ldots,l_n\}}]) \end{array}$$

In the last two rewriting examples and in the rules of replacement, a significant difference from the classic λ-calculus can be seen. In λ-calculus the multiple abstraction $\lambda a, b, c.Term$ is just a shorthand for $\lambda a.\lambda b.\lambda c.Term$, while in GENS these two expression have a different semantics. The reason for this is that the variable names in GENS exist outside of their lambda abstraction, so the renaming of free variables can not be done.

2.4 Sequence and field

Every language construct described so far has descended from the functional languages. The environment can be seen as a function call with its arguments named, and the lambda-abstraction is in the very basis of the most of functional programming languages. Imperative languages, on the other hand, are based on the control flow, or the instruction sequence. Therefore, in order to represent the imperative languages in a natural way in GENS, we have to extend it with some new constructs:

```
Term   = Label | Environment | Abstraction | Sequence | Field.
Sequence = Term ';' Term.
Field  = Term '.' Term.
```

We shall soon see that sequence and rewriting are actually just a syntactic sugar and that they can be represented through other constructs. But for now, let us consider them as new primitive constructs. Their behavior can be easily seen from some examples.

Example 4.

$$
\begin{array}{lcl}
a;b & \rightarrow^* & b \\
a(c=d);b & \rightarrow^* & b(c=d) \\
a(c=d);c & \rightarrow^* & d(c=d) \\
a(c=d);b(c=e) & \rightarrow^* & b(c=e) \\
a(c=d);b(e=c) & \rightarrow^* & b(c=d,e=c) \\
a(c=d);\backslash c- > b(e=c) & \rightarrow^* & b(c=d,e=d)
\end{array}
$$

$$
\begin{array}{lcl}
a.b & \rightarrow^* & b \\
a(c=d).b & \rightarrow^* & b \\
a(c=d).c & \rightarrow^* & d \\
a(c=d).b(c=e) & \rightarrow^* & b(c=e) \\
a(c=d).b(e=c) & \rightarrow^* & b(e=c) \\
a(c=d).\backslash c- > b(e=c) & \rightarrow^* & b(e=d)
\end{array}
$$

A sequence is a construct known from imperative languages. The rewriting of the sequence $Term1;Term2$ is done by reducing first $Term1$ to its root normal form $Term1'$. Then we take its mapping and proceed with rewriting of the environment $\langle Term2/\sigma_1' \rangle$. The multiple sequence

$$Term_1;Term_2;\ldots;Term_N$$

is grouped to the right, as $Term_1;(Term_2;(\ldots;Term_N)\ldots)$. Each subsequent term adds to the mapping passed from left to the right.

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

The rewriting rules defining semantics for a sequence and a field, are as follows (here $\perp$ denotes an undefined value).

$$
\begin{array}{rl}
T = & L \cup \left\{ \langle t/\sigma \rangle \,|t \in T \wedge \sigma : L \mapsto T \right\} \cup \left\{ \lambda \vec{l}.t | t \in T \wedge \vec{l} \in L^* \right\} \cup \\
& \cup \left\{ t_1;t_2 | t_1,t_2 \in T \right\} \cup \left\{ t_1.t_2 | t_1,t_2 \in T \right\}
\end{array}
$$

$$
\begin{array}{rcll}
\langle \langle t/\sigma_1 \rangle /\sigma_2 \rangle & \rightarrow & \langle t \,/\, \sigma_1 \triangleright \sigma_2 \rangle & \\
\langle l/\sigma \rangle & \rightarrow & \langle \sigma(l)/\sigma \rangle, & \text{if } \sigma(l) \neq \bot \\
\left\langle \lambda \vec{l}.t/\sigma \right\rangle & \rightarrow & \left\langle t[\vec{l} := \sigma(\vec{l})]/\sigma \right\rangle, & \text{if } (\forall i)(\sigma(l_i) \neq \bot) \\
 & & & \\
\langle t_1; t_2/\sigma \rangle & \rightarrow^* & t_2', & \text{if } \langle t_1/\sigma \rangle \rightarrow^* \langle t_1'/\sigma_1' \rangle \wedge \\
 & & & \wedge \langle t_2/\sigma_1' \rangle \rightarrow^* t_2' \\
\langle t_1.t_2/\sigma \rangle & \rightarrow^* & \langle t_2'/\sigma_2' \rangle, & \text{if } \langle t_1/\sigma \rangle \rightarrow^* \langle t_1'/\sigma_1' \rangle \wedge \\
 & & & \wedge \langle t_2 \,/\, \{\} : \sigma_1' \rangle \rightarrow^* \langle t_2' \,/\, \sigma_2' : \sigma_1' \rangle
\end{array}
$$

The given rules are sensitive to the order of application. We leave them that way, because the final semantics of GENS will be presented soon (in Section 2.6).

2.5 Failure and disjunction

The elements of GENS described so far are quite enough to describe not only a simple command sequence from imperative languages, but also the branching, loops and other control structures. However, the behavior of an exception is very hard to describe as an instruction sequence (it is practically impossible to extend an imperative language with exceptions without substantial modification of its semantics).

The modification in semantics of GENS needed to support exceptions, will be simple: we only need a name to denote error. This name, *Fail*, will have a special treatment in the semantics. If a term reduces to *Fail*, we shall say that the rewriting failed. Now, if the rewriting of the first term in the sequence $Term1; Term2$ fails, the rewriting is not continued to $Term2$ but stopped. The rewriting of a sequence results with *Fail*. The same holds for a field.

There still remains the problem of "catching" and correcting the exception. Hence, we add yet another (and last) construct to GENS: Disjunction of two terms.

```
Term        = Label | Environment | Abstraction |
              Sequence | Field | Disjunction.
Disjunction = Term '|' Term.
```

Its behavior is as follows: if $Term1$ reduces to any root normal form other than *Fail*, that is the root normal form of the whole disjunction. If rewriting of $Term1$ fails, then the normal form of disjunction is equal to the normal

form of the other term, $Term2$. A disjunction is used to correct the local failure and to finally reduce the whole expression successfully.

The required changes in the semantics of GENS can be seen from the following picture:

$$T = L \cup \{\langle t/\sigma\rangle \,|t \in T \wedge \sigma : L \mapsto T\} \cup \{\lambda\vec{l}.t|t \in T \wedge \vec{l} \in L^*\} \cup \\ \cup\{t_1;t_2|t_1,t_2 \in T\} \cup \{t_1.t_2|t_1,t_2 \in T\} \cup \{t_1|t_2\,|t_1,t_2 \in T\} \cup \{Fail\}$$

$$\begin{aligned} \langle\langle t/\sigma_1\rangle/\sigma_2\rangle &\to \langle t \,/\, \sigma_1 \triangleright \sigma_2\rangle \\ \langle l/\sigma\rangle &\to \langle\sigma(l)/\sigma\rangle, && \text{if } \sigma(l) \neq \bot \\ \langle\lambda\vec{l}.t/\sigma\rangle &\to \langle t[\vec{l} := \sigma(\vec{l})]/\sigma\rangle, && \text{if } (\forall i)(\sigma(l_i) \neq \bot) \end{aligned}$$

$$\begin{aligned} \langle t_1;t_2/\sigma\rangle &\to^* Fail, && \text{if } \langle t_1/\sigma\rangle \to^* Fail \\ \langle t_1;t_2/\sigma\rangle &\to^* t_2', && \text{if } \langle t_1/\sigma\rangle \to^* \langle t_1'/\sigma_1'\rangle \wedge \langle t_2/\sigma_1'\rangle \to^* t_2' \end{aligned}$$

$$\begin{aligned} \langle t_1.t_2/\sigma\rangle &\to^* Fail, && \text{if } \langle t_1/\sigma\rangle \to^* Fail \\ \langle t_1.t_2/\sigma\rangle &\to^* \langle t_2'/\sigma_2'\rangle, && \text{if } \langle t_1/\sigma\rangle \to^* \langle t_1'/\sigma_1'\rangle \wedge \\ & && \wedge\langle t_2 \,/\, \{\}:\sigma_1'\rangle \to^* \langle t_2' \,/\, \sigma_2':\sigma_1'\rangle \end{aligned}$$

$$\begin{aligned} \langle t_1|t_2/\sigma\rangle &\to^* t_1', && \text{if } \langle t_1/\sigma\rangle \to^* t_1' \neq Fail \\ \langle t_1|t_2/\sigma\rangle &\to^* t_2', && \text{if } \langle t_1/\sigma\rangle \to^* Fail \wedge \langle t_2/\sigma\rangle \to^* t_2' \end{aligned}$$

The given rules are also sensitive to the order of application. The final semantics will be given in the following section.

Example 5.

$$\begin{aligned} &Fail;a && \to^* Fail \\ &Fail.a && \to^* Fail \\ &a|b && \to^* a \\ &(a(b = Fail)|a(b = c));b && \to^* Fail \end{aligned}$$

One dubious feature of the presented semantics can be seen in the last example. Since $a(b = Fail)$, the first branch of the disjunction, succeeds (doesn't reduce to Fail), it is taken as the result of disjunction. When the sequence rewriting continues, it leads to the term $b(b = Fail)$ which fails. If we are allowed to "change our mind" in this moment and choose the other branch of disjunction, the final result would not be $Fail$ but $c(b = c)$. The rewriting would succeed even though a failure occurred after the disjunction rewriting. Then we would have a semantics similar to backtracking in Prolog.

2.6 Continuation

Continuation technique is commonly used in functional programs. It consists of transforming every function in such a way that it accepts one additional parameter, called continuation, which the function calls after preparing the environment. The biggest objection to this technique is that the transformed functions lose much in readability and become harder to use, since they need to receive yet another parameter. However, since all parameters in GENS can be inherited implicitly, the readability of the programs containing continuations is not of our concern.

What follows is the final GENS semantics.

$$\begin{aligned} T = \; & L \cup \{\langle t/\sigma\rangle \,|\, t \in T \wedge \sigma : L \mapsto T\} \cup \{\lambda\vec{l}.t | t \in T \wedge \vec{l} \in L^*\} \cup \\ & \cup \{t_1 | t_2 \,|\, t_1, t_2 \in T\} \cup \{Fail, Split, Drop\} \end{aligned}$$

$$\langle \langle t/\sigma_1\rangle / \sigma_2\rangle \;\rightarrow\; \langle t \,/\, \sigma_1 \triangleright \sigma_2\rangle$$

$$\langle l/\sigma\rangle \;\rightarrow\; \begin{cases} \langle \sigma(l)/\sigma\rangle, & \text{if } \sigma(l) \neq \bot \\ \langle C/\sigma\rangle, & \text{if } \sigma(l) = \bot \wedge \sigma(C) \neq \bot \end{cases}$$

$$\left\langle \lambda\vec{l}.t/\sigma \right\rangle \;\rightarrow\; \begin{cases} \left\langle t[\vec{l} := \sigma(\vec{l})]/\sigma \right\rangle, & \text{if } (\forall i)(\sigma(l_i) \neq \bot) \\ \langle C/\sigma\rangle, & \text{if } (\exists i)(\sigma(l_i) = \bot) \wedge \sigma(C) \neq \bot \end{cases}$$

$$\begin{aligned} a; b \;&\equiv\; \lambda C.\left\langle a/\left\{(C, \langle b/\{(C, C)\}\rangle)\right\}\right\rangle \\ a.b \;&\equiv\; Split; a; Split; b; Drop \\ \langle t_1 | t_2/\sigma\rangle \;&\rightarrow^*\; \begin{cases} t_1', & \text{if } \langle t_1/\sigma\rangle \rightarrow^* t_1' \neq Fail \\ t_2', & \text{if } \langle t_1/\sigma\rangle \rightarrow^* Fail \wedge \langle t_2/\sigma\rangle \rightarrow^* t_2' \end{cases} \\ \langle Split/\sigma\rangle \;&\rightarrow\; \langle C \,/\, \{\} : \sigma\rangle \\ \langle Drop/\sigma_1 : \sigma_2 : \sigma_3\rangle \;&\rightarrow\; \langle C \,/\, \sigma_1 \triangleright \sigma_3\rangle \\ \langle Fail/\sigma\rangle \;&\rightarrow\; Fail \end{aligned}$$

The continuation parameter will be passed over as the value of the special name, which is in the semantics definition denoted by C. If the continuation is empty, all terms behave as described by now. In the other case, any expression that reduces to the root normal form (except the name *Fail*) should not return the result until it reduces first the continuation.

What is gained by continuation? First, it allows us to define the sequence and field as common terms (so we "proved" the claim that sequences and fields in GENS are only syntactic sugar). The changes needed in the GENS

semantics are not too big. Second, we solve the problem of backtracking mentioned in the previous section. Now the rewriting of the last term succeeds even if a failure happens in its continuation, though the disjunction has already been successfully reduced:

$$(a(b = Fail)|a(b = c)); b \rightarrow^* c(b = c).$$

2.7 Atoms

Every programming language has some built-in basic data types. In GENS these types are called atoms and they include 16-bit integers, characters, strings and texts (textual files). This choice has been made according to the basic purpose of the language, to experiment in the programming language area.

There is not much to say about integer atoms. Character atoms are written between the single quotes, and strings can be written between either single or double quotes.

Atoms can not be rewritten, which means they are always in normal form. Contrary to labels, they cannot be assigned a value by any environment and they cannot be in any environment root. The only way to handle atoms is through the built-in functions, which shall be described in the following section.

2.8 Primitive functions

Every built-in operation is assigned to one label in the system environment. There are also predefined names that are not assigned any value, but are used as the argument-holders for built-in operations. These are for example `Value`, `Property`, `Test`, `Yes`, `No`, etc. The argument names are fixed.

All built-in operations of GENS can be roughly divided in three groups: low-level operations, operations on atoms, and parser operations. We shall shortly describe here only those functions that will be mentioned in the rest of the paper.

Low level operations are used for elementary changes of the environment.

`Seq`, `Field` and `Dis` are the root names of sequence, field and disjunction. For example, the term $Seq(Left = Term1, Right = Term2)$ is equivalent to the sequence $Term1; Term2$. `Reduce` reduces the value of the name Value

to its root normal form. `Set` assigns the value of the name `Value` to the name that is the value of the name `Property`. `Lambda` is defined as follows: $Lambda(Property = X, Value = Term) \equiv \backslash X -> Term$. `FreshVariable` creates a new name and assigns it to the label `Value`. `Guard` and `Match` are the labels of the operations used for pattern matching and unification. `System` holds the system environment, which defines all the operations given here.

Operations on atoms are used to work with atoms.

`Add`, `Sub`, `Mul`, and `Div` are the functions for adding, subtracting, multiplying and dividing integers. The expected parameters are the values of labels `1st` and `2nd`. Before the operation the values of these names are reduced to root normal form, which means that these functions are strict. `Equal`, `Less`, `Greater`, `LessEq`, `GrEq`, and `Different` are predicates which compare the value of the name `1st` with the value of the name `2nd`. In case the relation is satisfied the result is `True`, and otherwise the rewriting fails. `And`, `Or` and `Not` are the standard logical operations. For example, `And` succeeds with the result `True` if the rewriting of both names `1st` and `2nd` succeeds. `If` is the primitive branching function. If the rewriting of the name `Test` succeeds, the name `Yes` reduces, and otherwise name `No`.

Names like `1st`, `2nd`, `3rd` etc. are often used when we do not have a more sensible name for an argument. Therefore these names are taken for default wherever the label of the attribute is left out. For example, the term `a(b, 7, x=c)` is equivalent to the term `a(1st= b, 2nd= 7, x= c)`.

2.9 References

In many modern procedural languages the parameter passing through a reference is used, a mechanism hard to combine into declarative programming languages.

When we use explicit references we must explicitly state when we want to access the variable that holds a reference, and when the variable it refers to, for example `a:= b` and `a^ := b` in Pascal. With implicit references, the two variables become complete synonyms, so that by changing the value of one variable we can change the other without knowing it. An example of implicit references is the parameter passing by a reference from Pascal and its descendants.

Usefulness and sensibility of references in declarative programming lan-

guages can be discussed. However, since GENS is not created as a general-purpose language, but as a meta-language for implementation of a wide class of other languages, a reference mechanism has been added to it.

Explicit references can be modeled with no need for a new extension of GENS. The name `Transparent` is used to model implicit references. This operation takes the label that is the value of `Property`, and makes it implicitly refer to its former value. If `Property`'s value is not a name, the rewriting fails.

$$\begin{array}{ll} Transparent(Property = x, x = a); & \\ Let(a = 1); & \\ Let(x = 2) & \rightarrow^* \quad Let(a = 2, x = 2) \end{array}$$

3. Conclusion

GENS has been implemented in the programming language Oberon-2 under Oberon operating system. It has been used for implementation of several programming languages: ISWIM (functional language), subset of Prolog (logical language), Pascal (subset of a procedural language) and their various combinations. The implementation of a simple object-oriented language is currently underway.

GENS is a low-level language. To an unexperienced programmer, GENS programs are hard to read and write. To ease the usage of GENS, a specialized language G has been developed. G is a language similar to BNF and to languages used in compiler generators, which use attributed grammars to specify syntax and semantics of (other) programming languages. G has been written in GENS and its purpose is to define syntax of a programming language (or arbitrary term) and to specify its translation into an equivalent GENS program.

Using G, the mentioned programming languages have been implemented. Besides, lambda calculus and "classical" graph rewriting system have been implemented as well. All implemented languages can be used for implementation of other languages thus forming generations of languages emerging from GENS.

The translation of a programming language (say P) is fully automated. When a P program is loaded into the system, its translation down the chain of languages is automatically invoked. The parser created for a language P is then saved so that every other translation is done directly.

This is achieved by considering filenames as attribute labels, file contents as attribute values and the set of files in a folder as the environment.

In this way GENS has been extended to an integrated programming environment for programming language development. Further research will be concentrated onto the support for persistence, input/output and other issues necessary for the development of practical programming languages.

References

[1] Aït-Kaci, H. and Garrigue, J., Label-selective lambda-calculus: syntax and confluence, in: Proceedings of the 13th Int. Conf. on Foundations of Software Tech. and Theoret. Comput. Sci., Lecture Notes in Comput. Sci., Vol. 761, pp. 24–40, Springer-Verlag, 1993.

[2] Barendregt, H., van Eekelen, M., Glauert, J., Kennaway, J., Plasmeijer, M. and Sleep, M., LEAN - an intermediate language based on graph rewriting, Parallel Comput. 9 (1988), 163–177.

[3] Blažević, M. and Budimac, Z., Attributed graph rewriting system, in: Proc. XII Conf. on Appl. Math. PRIM '97. (Subotica, 1997), pp. 11–19, University of Novi Sad, 1998.

[4] Blažević, M., Reduction of attributed graphs (in Serbian), in: Proc. Conf. ETRAN '98. (Zlatibor, 1998), to appear.

[5] Blažević, M., Implementing Prolog using attributed graph reduction (in Serbian), Seminar paper, 45 pp., Institute of Mathematics, University of Novi Sad, 1998.

[6] Budd, T. A., Multiparadigm Programming in Leda, Addison-Wesley, 1995.

[7] Dami, L., A lambda-calculus for dynamic binding, Theoret. Comput. Sci. 192 (1998), 201–231.

[8] Dami, L., A comparison of record- and name-calculi, in: ed. D. Tsichritzis, "Objects at large", pp. 71–83, Centre Universitaire d'Informatique, University of Geneva, 1997.

[9] van Eekelen, M., Huitema, H., Nöcker, E., Plasmeijer, M. and Smetsers, J., Concurrent Clean, Language Manual 0.8, Technical Report 92-18, 50 pp., University of Nijmegen, 1992.

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

[11] Hashimoto, M. and Ohori, A., A typed context calculus, Internet homepage of Kyoto University, Japan.

[12] Moreno Navarro, J. J., Expressivity of functional-logic languages and their implementation, manuscript, 32 pp.

[13] Papadopoulos, G., Concurrent object-oriented programming using term graph rewriting techniques, Inform. and Software Tech. 38 (1996), 539–547.

[14] Peyton Jones, S. and Wadler, P., Imperative functional programming, in: Proc. 20th ACM Symp. on Principles of Programming Languages (Charleston, 1993), pp. 71–84, Assoc. Comput. Mach., 1993.

Received March 19, 1999.