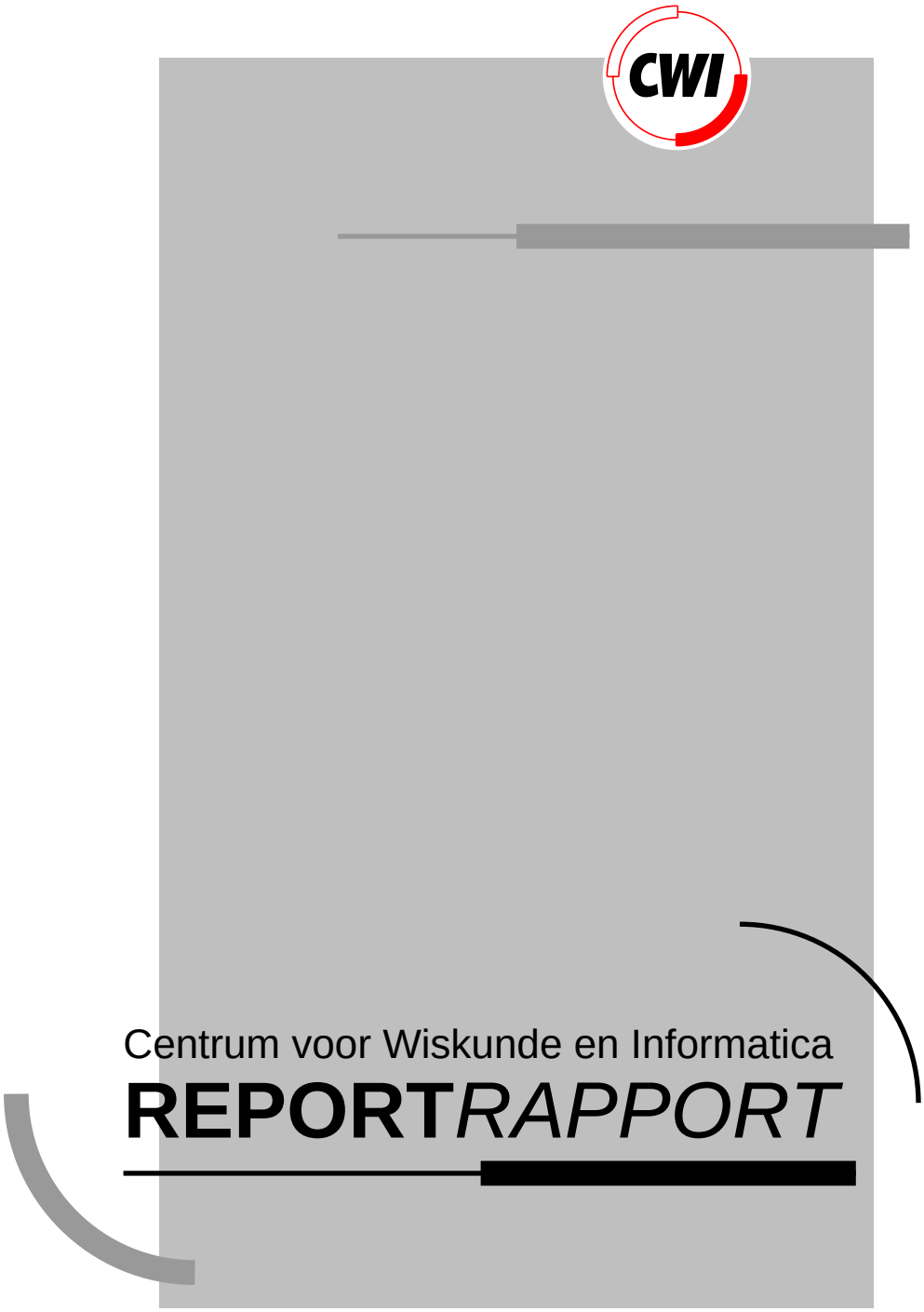


Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

GEL, a Graph Exchange Language

J.F.T. Kamperman

Computer Science/Department of Software Technology

CS-R9440 1994

GEL, a Graph Exchange Language

J.F.Th.Kamperman (jasper@cwi.nl)

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

GEL (Graph Exchange Language) is a formalism for the compressed exchange of (term) graphs between processes. Key features of GEL are speed, compactness, independence of readers and writers, and compositionality. Typically, GEL representations of large, tree-like graph structures require an average of a little more than one byte storage for representing one node in the graph.

Orthogonally to GEL, other protocols can be used to exchange data residing in the nodes of a graph. An algebraic specification of the semantics of GEL texts is given, as well as performance measurents of an experimental C implementation.

The C implementation can be ftp'ed at ftp.cwi.nl as pub/gipe/sources/GEL.tar.Z.

CR Subject Classification (1991): C.2.0 `[Computer-communication networks]`: Data communications; C.2.4 `[Computer-communication networks]`: Distributed systems; D.2.1 `[Software Engineering]`: Requirements/Specifications; D.2.2 `[Software Engineering]`: Tools and techniques; D.2.6 `[Software Engineering]`: Programming Environments; D.2.m `[Miscellaneous]`: Rapid prototyping, reusable software; D.3.4 `[Programming languages]`: Processors; E.2 `[Data Storage Representations]`: Composite structures; E.4 `[Coding and Information Theory]`: Data compaction and compression, nonsecret encoding schemes and G.2.2 `[Mathematics of Computing]`: Graph algorithms.

AMS Subject Classification (1991): 68N15 [*Software*]: Programming Languages; 68N20 [*Software*]: Compilers and generators; 68Q65 [*Theory of Computing*]: Algebraic specification and 90C35 [*Mathematical Programming*]: Programming involving graphs.

Keywords & Phrases: graphs, trees, exchange formats, compression, distributed systems, algebraic specification, prototyping, software development methodology, C.

Note: Partial support received from the European Communities under `ESPRIT` project 5399 (Compiler Generation for Parallel Machines – `COMPARE`). This report also appeared as deliverable D3.21.1/2.

1. Introduction

Graph-structured data types play a role in a large variety of complex software systems. Especially software performing symbolic manipulation, such as a compiler or a symbolic algebra system, makes use of this kind of data type. Several trends, e.g. the growth of distributed computing and the integration of software developed for complementary purposes, cause a demand for the efficient, language-independent, exchange of graph-structured data.

There are several approaches in existence for the exchange of data, independent of any implementation language, notably ASN.1 (Abstract Syntax Notation One, [CCI]) and the

ASCII external representation (ERL) of IDL (Interface Definition Language, [Sno89]). Of these formalisms, GEL (Graph Exchange Language) bears most similarity in functionality to ERL. For every node in a graph, an ERL description gives a type, a sequence of named edges with corresponding subgraphs, and possibly a label for non-local references.

So why another formalism for the exchange of data? GEL can be characterized by observing how it extends the capabilities of the ERL representation:

- GEL contains a dynamic abbreviation mechanism, which allows the use of very short identifiers in the bulk of the text.
- Instead of labels to identify shared subgraphs or circularities, GEL has relative indices. This enables faster access of shared subgraphs.
- The semantics of GEL assumes the existence of a stack of subgraphs, leading to an efficient implementation for the important class of DAGs (Directed Acyclic Graphs). In [DR94], a message protocol is defined that uses an implicit stack of subtrees in a similar way as GEL, though no compression, sharing or circularity is supported.
- GEL is compositional: if a graph is composed of several subgraphs, its GEL text can be composed of the GEL texts of its subgraphs. In ERL, the labels in the texts of the subgraphs have to be made unique before composing.

We claim that in many cases, GEL removes the need to implement a more efficient exchange protocol for production versions of a system ([Sno89], page 141). Thus, components in the prototype phase can be mixed freely with production versions of other components.

GEL is more austere than ERL, because there are no built-in datatypes `string` or `integer`. At reasonable costs, these datatypes can be implemented on top of GEL. GEL deals exclusively with the compressed exchange of graph-structured data.

In the ASF+SDF meta-environment [Kli93] for the generation of programming environments from algebraic specifications in ASF+SDF [BHK89a], (term) graphs are the basic data structure. Currently, this system is migrating from a monolithical implementation in Lisp into a distributed implementation, which is itself partly generated from an ASF+SDF specification [BK94]. GEL is used satisfyingly for the exchange of terms between the generated distributed components.

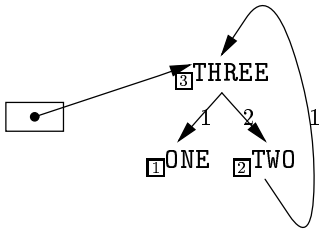
We will first give an informal overview of GEL in Section 2. In Sections 4 to 8, we will present the full GEL language by way of a guided tour through an algebraic specification of the GEL reader in the ASF+SDF formalism, which is briefly introduced in Section 3.

In Section 9, we present an algorithm for writing GEL texts. In Section 11, we go into the design decisions that led to the ultimate design of GEL, and in Section 12, we present timings of our experimental C implementation.

In Appendices A and B, we give a binary encoding of GEL, and the interface description of our implementation as a library for use with the `C` programming language [KR78].

2. An informal overview of GEL

GEL describes rooted, directed, connected graphs with typed (labeled) nodes and ordered edges, called *term graphs* in [BvEJ+87]. Most graph-structured datatypes can easily be




```
! a : 0 = ONE <RET>
a <RET>
! b : 1 = TWO <RET>
> 1 <RET>
b <RET>
! c : 2 = THREE <RET>
c <RET>
```

Figure 1: A rooted graph and its GEL encoding

mapped to GEL graphs.

As an informal introduction, we have drawn a suggestive picture of a GEL graph in Figure 1. The rectangular box with the arrow is a pointer to the root of the graph. The types of the nodes in this graph are `ONE`, `TWO` and `THREE`. For easy reference, the nodes are numbered in their lower left corner. The direction of an edge is indicated by its arrowhead, and the ordering of the edges is indicated by numbers (1,2).

Basically, a GEL text defines abbreviations for the compression of types, and gives directives to build nodes with edges pointing to other nodes. GEL is a stack-based language, i.e., the directive to build a node assumes that the nodes referred to are on top of a stack in the appropriate order. This allows for a very short encoding (postfix) in case the graphs are trees. If the graphs fall in a more general class, special elements are pushed onto the stack, taking care of multiple references to the same node, or even cyclic references.

As a short example, consider the GEL text in Figure 1. The lines in this figure are a somewhat more readable representation of the actual binary encoding of GEL defined in Appendix A. The first line in this figure defines an abbreviation `a` for the type `ONE`, which has no edges. The second line builds a node of this type, and puts it on the stack. The third line defines an abbreviation `b` for the type `TWO`, which has one edge. The fourth line puts a future reference on the stack, which is used in the fifth line to build a node of the type `TWO`, with one edge. The fifth line defines an abbreviation for the type `THREE`, which has two edges. This abbreviation is used in the sixth line to build a node with two edges. The last reference remaining on the stack is interpreted as the root of the graph, so the GEL text in the right part of Figure 1 describes the graph in the left part of Figure 1.

3. A quick overview of ASF+SDF

ASF+SDF is a specification formalism for describing all syntactic and semantic aspects of (formal) languages. It is an amalgamation of the formalisms SDF [HHKR89] for describing syntax, and ASF [BHK89b] for describing semantics.

ASF is a conventional algebraic specification formalism providing notions like first-order signatures, import/export, variables, and conditional equations. The meaning of ASF specifications is based on their initial algebra semantics. If specifications satisfy certain criteria, they can be executed as a term rewriting system.

SDF introduces the idea of a "syntactic front-end" for terms and equations defined over a

first-order signature. This creates the possibility to write first-order terms as well as equations in arbitrary concrete syntactic forms: from a given SDF definition for some context-free grammar, a fixed mapping from strings to terms can be derived.

An ASF+SDF specification consists of a sequence of named modules. Each module may contain:

Imports of other modules.

Sort declarations defining the sorts of a signature.

Lexical syntax defining layout conventions and lexical tokens.

Context-free syntax defining the concrete syntactic forms of the functions in the signature.

Variables to be used in equations. In general, each variable declaration has the form of a regular expression and defines the class of all variables whose name is described by the regular expression.

Equations define the meaning of the functions defined in the context-free syntax section.

An unusual feature, associative *lists* will be used in this paper, and deserves some further explanation. In the description of context-free grammars one frequently encounters the notion of iteration or list, in order to describe syntactic constructs like statement-list, parameter-list, declaration-list, etc. In ASF+SDF this notion is provided at the syntactic as well as at the semantic level. At the syntactic level, one can define, for instance, a "list of zero or more identifiers separated by comma's". At the semantic level, variables over such lists may be declared and used in equations. Semantically, lists can always be eliminated. Operationally, the matching of a list structure is achieved by local backtracking during term rewriting [Hen89].

For the presentation of our specification, we use Eelco Vissers "TOLATEX" package. Apart from providing a nice layout for terms and equations, it prints section numbers in the upper right corner of imports, for easy reference.

4. An overview of the specification of GEL

In Figure 2, the structure of the GEL specification is displayed. Starting at the bottom, and proceeding from left to right, we have the following modules:

Ints Integers are used to number the nodes and edges in a graph. The specification of this module can be found in [Wal94].

Layout Layout is needed in order to present the equations of modules in a readable way. However, the GEL formalism is very restrictive with respect to layout. Therefore, much care is taken to only import Layout where necessary. Because this module is not specific to GEL, it is given in Appendix **??**.

Gel-types Specifies basic notions which occur both in graphs and in their GEL descriptions. Section 5.1 explains the module.

Graphs In this module the subject matter of GEL is defined formally. A discussion can be found in Section 5.2.

Gel Here, the syntax of commands occurring in GEL is defined. See Section 7.1 for an overview of the commands. The full lexical definition of sorts is deferred to Full-Gel-syntax (see below).

Stacks During the construction of graphs, stacks of subgraphs are used. These stacks and the operations on them are defined in this module. Section 6.2 gives more detail.

Tables While reading a GEL text, the GEL reader must maintain a correspondence between abbreviated and full type names. The datatype needed for this is specified in the module Tables. Section 6.1 contains the full specification.

Full-Gel-syntax The full definition of the readable form of GEL. Ideally, this definition should be given by the module Gel, but this would cause parsing problems in the equations of Gel-read. In Section **??**, these subtleties are explained.

Gel-read Finally, the effect of the commands available in GEL is defined by a specification of the transitions of an abstract machine. Given the auxiliary functions defined in the preceding modules, every command can be specified in a single equation. Section 7.2 elaborates on this module.

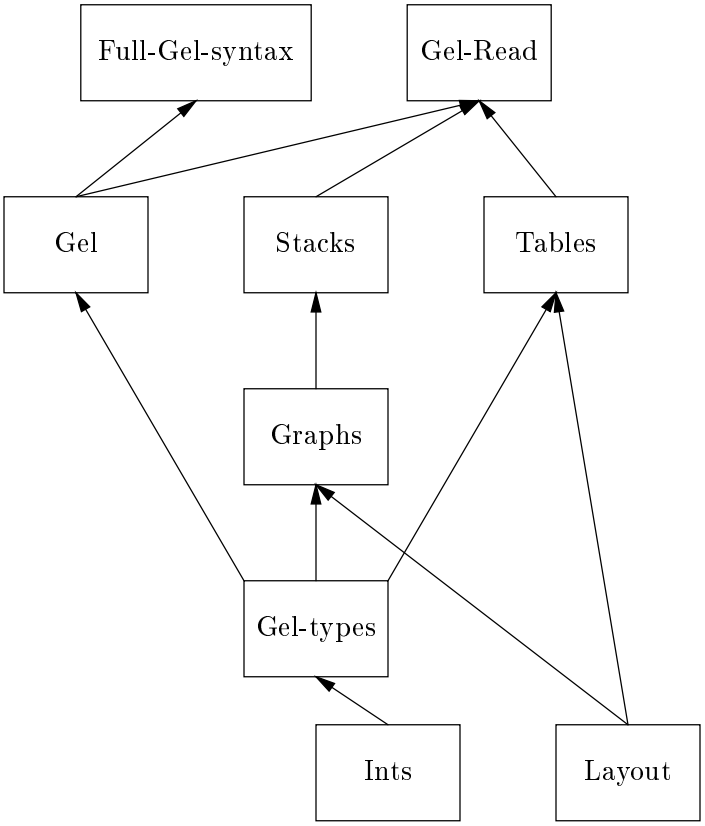


Figure 2 The structure of the GEL specification

5. Basics: Gel-types and Graphs

5.1 Gel-types

Module Gel-types

imports Ints(13)
exports
sorts TYPE SHORT-TYPE RET SPACE
lexical syntax
[′0-9a-zA-Z]+ → TYPE
[′a-zA-Z][a-zA-Z0-9]∗ → SHORT-TYPE
context-free syntax
"∗" → INT
variables
Type [0-9′]∗→ TYPE
Space → SPACE
Ret → RET
hiddens
lexical syntax
[␣\t\n] → LAYOUT
"//"∼[\n]∗[\n] → LAYOUT
"%%"∼[\n]∗[\n] → LAYOUT
equations

∗ = − 1 [varyad]

As far as GEL is concerned, the possible types of nodes are an uninterpreted parameter. In the main part of our specification, we model this by a lexical definition of the sort TYPE that admits identifiers with quotes. Only in the module Full-Gel-syntax, this is extended to any string that does not contain a newline. In the binary implementation of GEL (see Appendix A), a TYPE can be any (length-encoded) sequence of bytes.

In GEL, abbreviations can be introduced for types, where the lexical syntax for abbreviations is given by the sort SHORT-TYPE. In the binary implementation of GEL, a SHORT-TYPE is replaced by a binary encoding of a number (index in a table).

In order to parse the equations without exporting lexical syntax for LAYOUT, LAYOUT is defined locally.

In order to stress the non-numeric meaning of −1 (see section 5.2), we introduce an alias ∗.

5.2 Graphs described by GEL
Module Graphs
imports Ints(13) Gel-types(5.1) Layout(D)
exports
sorts EDGE EDGES NODE NODES GRAPH INTS
context-free syntax
node INT *of type* TYPE *with sig* INT *and edges* EDGES → NODE
"[" {NODE ","}∗ "]" → NODES

root is INT *in* NODES → GRAPH
error_graph → GRAPH

INT "→" INT → EDGE

"{" {EDGE ","}* "}" → EDGES

variables

Edge [0-9′]* → EDGE
Edge [0-9′]*"*" → {EDGE ","}*
Graph [0-9′]* → GRAPH
Node [0-9′]* → NODE
Node [0-9′]*"*" → {NODE ","}*

hiddens

context-free syntax

occurs INT *in* NODES → INT
occurs INT *in* EDGES → INT
"{" {INT ","}* "}" → INTS
occurs INT *in* INTS → INT
NODES *close with* INTS → INTS
num-edges EDGES → INT

variables

Int [0-9′]*"*" → {INT ","}*

In the specification of GEL's graphs, the nodes are taken as the basis of description. A node has a number (INT) for identification purposes, a type (TYPE) which is uninterpreted in our specification, an arity indication (INT) and a number (possibly zero) of edges (EDGE). The arity indication is redundant for the specification of the graph structure, but it enables a more efficient GEL representation (see Section 11). Non-negative arities specify the number of edges, an arity of `-1` specifies a *varyadic* arity, meaning that the node can have any number of edges. A GEL graph consists of a number of nodes, with a (positive, see below) integer indicating the root.

We have left unspecified, that a consistent renumbering of the nodes and the root (cf. alpha-conversion in the λ-calculus) yields exactly the same graph structure. Finally, an edge is a pair of integers (INT), where the first integer indicates the node of departure, and the second integer indicates the destination node. It is slightly awkward that integers play three different roles (node identifier, arity indication and edge label). We did not specify three different sorts for these roles, because integer arithmetic is convenient in all three cases. Given these definitions, we present in Figure 3 a term representation of the GEL graph in Figure 1.

root is 3
in [*node* 1 *of type* `ONE` *with sig* 0 *and edges* {},
  node 2 *of type* `TWO` *with sig* 1 *and edges* {1 → 3},
  node 3 *of type* `THREE`
  with sig 2
  and edges {1 → 2,
    2 → 3}]

Figure 3 Term representation of the graph in Figure 1

equations

One element of the sort GRAPH is used as an error element:

error_graph = *root is* 0 *in* [] [error-graph-0]

Node numbers are positive integers

$$\frac{X_4 < 1 = 1}{\begin{array}{l}\textit{root is } X \textit{ in } [Node^{*\prime},\\ \qquad \textit{node } X_4 \textit{ of type Type with sig } X_2\\ \qquad \textit{and edges } \{Edge^*\},\\ \qquad Node^{*\prime\prime}]\end{array} = \textit{error\_graph}} \quad \text{[error-1]}$$

The root of the graph should point into the graph

$$\frac{\textit{occurs } X \textit{ in } [Node^*] = 0}{\textit{root is } X \textit{ in } [Node^*] = \textit{error\_graph}} \quad \text{[error-2]}$$

All edges should point into the graph

$$\frac{\begin{array}{l}[Node^*] = [Node^{*\prime},\\ \qquad \textit{node } X_4 \textit{ of type Type with sig } X_2\\ \qquad \textit{and edges } \{Edge^*,\ X_3 \to X_1,\ Edge^{*\prime}\},\\ \qquad Node^{*\prime\prime}],\\ \textit{occurs } X_1 \textit{ in } [Node^*] = 0\end{array}}{\textit{root is } X \textit{ in } [Node^*] = \textit{error\_graph}} \quad \text{[error-3]}$$

The number of edges should correspond to the signature

$$\frac{\begin{array}{l}[Node^*] = [Node^{*\prime},\\ \qquad \textit{node } X_4 \textit{ of type Type with sig } X_2\\ \qquad \textit{and edges } \{Edge^*\},\\ \qquad Node^{*\prime\prime}],\\ X_2 < 0 = 0,\\ \textit{num-edges } \{Edge^*\} \neq X_2\end{array}}{\textit{root is } X \textit{ in } [Node^*] = \textit{error\_graph}} \quad \text{[error-4]}$$

Every node should have a unique id

$$\begin{array}{l}\textit{root is } X_1 \textit{ in } [Node^*,\\ \qquad \textit{node } X \textit{ of type Type with sig } X_2 \textit{ and edges } \{Edge^*\},\\ \qquad Node^{*\prime},\\ \qquad \textit{node } X \textit{ of type Type}' \textit{ with sig } X_2{}' \textit{ and edges } \{Edge^{*\prime}\},\\ \qquad Node^{*\prime\prime}]\end{array} = \textit{error\_graph} \quad \text{[error-5]}$$

Every node should be reachable from the root

$$\frac{\begin{array}{l}[Node^*] \textit{ close with } \{X\} = \{Int^*\},\\ \qquad [Node^*] = [Node^{*\prime},\\ \qquad\qquad \textit{node } X_1 \textit{ of type Type with sig } X_2\\ \qquad\qquad \textit{and edges } \{Edge^*\},\\ \qquad\qquad Node^{*\prime\prime}],\\ \quad \textit{occurs } X_1 \textit{ in } \{Int^*\} = 0\end{array}}{\textit{root is } X \textit{ in } [Node^*] = \textit{error\_graph}} \quad \text{[error-6]}$$

Labels start from 1

$$\frac{X_1 < 1 = 1}{\begin{array}{l}\mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^*,\\ \quad \mathit{node}\ X_1\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_2\\ \quad \mathit{and\ edges}\ \{\mathit{Edge}^*,\ X_1 \to {X_1}',\ \mathit{Edge}^{*\prime}\},\\ \quad \mathit{Node}^{*\prime}] \quad =\\ \mathit{error\_graph}\end{array}} \quad [\text{error-7}]$$

Every label should occur only once on a node

$$\begin{array}{l}\mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^*,\\ \quad \mathit{node}\ X_1\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_2\\ \quad \mathit{and\ edges}\ \{\mathit{Edge}^*,\ X_1 \to {X_1}',\ \mathit{Edge}^{*\prime},\ X_1 \to {X_1}'',\ \mathit{Edge}^{*\prime\prime}\},\\ \quad \mathit{Node}^{*\prime}] \quad =\\ \mathit{error\_graph}\end{array} \quad [\text{error-8}]$$

Labels should be contiguous

$$\frac{\begin{array}{c}X_1 < 2 = 0,\\ \mathit{occurs}\ X_1 - 1\ \mathit{in}\ \{\mathit{Edge}^*,\ \mathit{Edge}^{*\prime}\} = 0\end{array}}{\begin{array}{l}\mathit{root\ is}\ X\ \mathit{in}\ [\mathit{Node}^*,\\ \quad \mathit{node}\ X_1\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_2\\ \quad \mathit{and\ edges}\ \{\mathit{Edge}^*,\ X_1 \to {X_1}',\ \mathit{Edge}^{*\prime}\},\\ \quad \mathit{Node}^{*\prime}] \quad =\\ \mathit{error\_graph}\end{array}} \quad [\text{error-9}]$$

Auxiliary functions define occurence in sets and closure under traversal

$$\frac{\mathit{occurs}\ X_3\ \mathit{in}\ \{\mathit{Int}^*,\ X,\ \mathit{Int}^{*\prime}\} = 0}{\begin{array}{l}[\mathit{Node}^*,\\ \ \mathit{node}\ X\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_1\\ \ \mathit{and\ edges}\ \{\mathit{Edge}^*,\ X_2 \to X_3,\ \mathit{Edge}^{*\prime}\},\\ \ \mathit{Node}^{*\prime}]\ \mathit{close\ with}\ \{\mathit{Int}^*,\ X,\ \mathit{Int}^{*\prime}\} \quad =\\ [\mathit{Node}^*,\\ \ \mathit{node}\ X\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_1\\ \ \mathit{and\ edges}\ \{\mathit{Edge}^*,\ X_2 \to X_3,\ \mathit{Edge}^{*\prime}\},\\ \ \mathit{Node}^{*\prime}]\ \mathit{close\ with}\ \{\mathit{Int}^*,\ X,\ X_3,\ \mathit{Int}^{*\prime}\}\end{array}} \quad [\text{closure}]$$

$$[\mathit{Node}^*]\ \mathit{close\ with}\ \{\mathit{Int}^*\} = \{\mathit{Int}^*\} \quad \textbf{otherwise} \quad [\text{closure}]$$

$$\mathit{occurs}\ X\ \mathit{in}\ \{\mathit{Int}^*,\ X,\ \mathit{Int}^{*\prime}\} = 1 \quad [\text{occurs-ints-0}]$$

$$\mathit{occurs}\ X\ \mathit{in}\ \{\mathit{Int}^*\} = 0 \quad \textbf{otherwise} \quad [\text{occurs-ints}]$$

$$\mathit{occurs}\ X\ \mathit{in}\ [\mathit{Node}^*,\ \mathit{node}\ X\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_1\ \mathit{and\ edges}\ \{\mathit{Edge}^*\},\ \mathit{Node}^{*\prime}] = 1 \quad [\text{occurs-0}]$$

$$\mathit{occurs}\ X\ \mathit{in}\ [\mathit{Node}^*,\ \mathit{node}\ X_1\ \mathit{of\ type}\ \mathit{Type}\ \mathit{with\ sig}\ X_2\ \mathit{and\ edges}\ \{\mathit{Edge}^*\},\ \mathit{Node}^{*\prime}] = 0 \quad \textbf{otherwise} \quad [\text{occurs}]$$

$$num\text{-}edges\,\{\} = 0 \qquad \text{[num-edges-0]}$$

$$num\text{-}edges\,\{Edge, Edge^*\} = 1 + num\text{-}edges\,\{Edge^*\} \qquad \text{[num-edges-1]}$$

6. The machine components

In Sections 6.1 and 6.2, we will specify the components of the GEL abstract machine. Here, we first give an informal overview. As a visual aid, a state of the GEL reading machine is depicted in Figure 4. A state consists of an abbreviation table, a stack of graph references, and an (unfinished) graph.

SHORT-TYPE	SIG	TYPE
a	2	APPLY
b	0	ONE
c	-1	LIST
d	2	PLUS

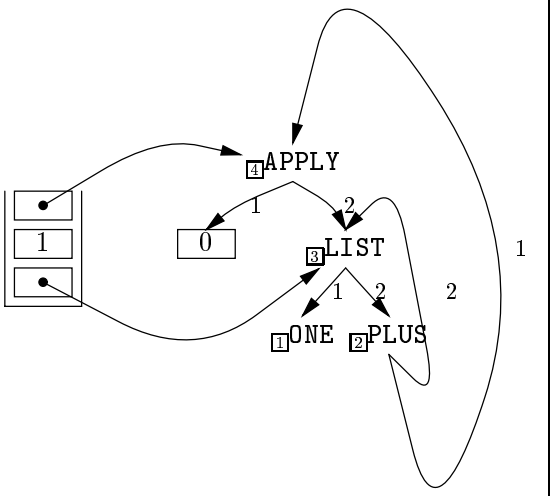


Figure 4 A GEL graph under construction

At the top of the figure, the abbreviation table is shown, associating an arity and a type with every abbreviation. For example, the abbreviation `a` is associated with a signature indicating an arity of 2, and a type `APPLY`.

Below the abbreviation table, on the left, the stack of graph references is shown, and on the right, we find the graph under construction. The nodes in the graph are shown as a type, tagged with a unique node number (only for reference, graphs are equivalent under node renumberings). Formally, the arity should also be indicated, but in our examples it follows from the context. The edges in the graph are arrows, tagged with their index in the edge-ordering.

The stack contains ordinary references and 'future' references. The ordinary references are depicted as rectangles from which arrows emanate, in this example to the subgraphs rooted at `APPLY` and `LIST`. The 'future' references are depicted as rectangles containing numbers,

indicating how far in the future the graph will be built. There is a future reference with number "1" on the stack, and a future reference with number "0" in the graph. The number in a future reference specifies the number of build operations to be performed before the indicated graph is constructed.

6.1 Tables

Module Tables
imports Layout$^{(D)}$ Gel-types$^{(5.1)}$
exports
sorts TABLE ITEM
context-free syntax
sig of SHORT-TYPE *in* TABLE → INT
type of SHORT-TYPE *in* TABLE → TYPE
define SHORT-TYPE *of type* TYPE *with sig* INT *in* TABLE → TABLE
empty-table → TABLE
variables
Table [0-9′]* → TABLE
Short-Type [0-9′]* → SHORT-TYPE

The concrete functions for the table should be hidden, but then the example terms further in the text become unparseable.

context-free syntax
abbr SHORT-TYPE *of type* TYPE *with sig* INT → ITEM
"[" {ITEM ","}* "]" → TABLE
variables
Item [0-9′]* → ITEM
Item [0-9′]*"*" → {ITEM ","}*
equations

$$empty\text{-}table = [] \qquad \text{[empty-table-0]}$$

$$define\ Short\text{-}Type\ of\ type\ Type\ with\ sig\ X\ in\ [Item^*] = [abbr\ Short\text{-}Type\ of\ type\ Type\ with\ sig\ X,\ Item^*] \qquad \text{[define-0]}$$

$$sig\ of\ Short\text{-}Type\ in\ [abbr\ Short\text{-}Type\ of\ type\ Type\ with\ sig\ X,\ Item^*] = X \qquad \text{[sig-0]}$$

$$\frac{Short\text{-}Type \neq Short\text{-}Type'}{sig\ of\ Short\text{-}Type\ in\ [abbr\ Short\text{-}Type'\ of\ type\ Type\ with\ sig\ X,\ Item^*] = sig\ of\ Short\text{-}Type\ in\ [Item^*]} \qquad \text{[sig-1]}$$

$$sig\ of\ Short\text{-}Type\ in\ Table = -2 \quad \textbf{otherwise} \qquad \text{[sig-error]}$$

$$type\ of\ Short\text{-}Type\ in\ [abbr\ Short\text{-}Type\ of\ type\ Type\ with\ sig\ X,\ Item^*] = Type \qquad \text{[type-0]}$$

$$\frac{Short\text{-}Type \neq Short\text{-}Type'}{type\ of\ Short\text{-}Type\ in\ [abbr\ Short\text{-}Type'\ of\ type\ Type\ with\ sig\ X,\ Item^*] = type\ of\ Short\text{-}Type\ in\ [Item^*]} \qquad \text{[type-1]}$$

$$type\ of\ Short\text{-}Type\ in\ [] = \texttt{error} \qquad \text{[type-2-error]}$$

A new abbreviation can be defined by *define*. It is possible that a certain SHORT-TYPE is defined several times. However, the equations for the access functions specify that only the last definition matters.

Given a SHORT-TYPE, the associated TYPE and SIG can be found by applying the functions *type of* and *sig of*.

6.2 Stacks

Module Stacks
imports Graphs$^{(5.2)}$
exports
sorts STACK
context-free syntax

empty	$\rightarrow$ STACK
push(INT, STACK)	$\rightarrow$ STACK
top(STACK)	$\rightarrow$ INT
top(INT, STACK)	$\rightarrow$ INT
pop(STACK)	$\rightarrow$ STACK
pop(INT, STACK)	$\rightarrow$ STACK
drop(INT, STACK)	$\rightarrow$ STACK
take(INT, STACK)	$\rightarrow$ EDGES

variables
Stack [*0-9'*]$*\rightarrow$ STACK
equations

$$top(push(X, Stack)) = X \qquad \text{[top-0]}$$

$$top(empty) = 0 \qquad \text{[top-error]}$$

$$top(0, Stack) = top(Stack) \qquad \text{[top-1]}$$

$$\frac{X < 1 = 0}{top(X, Stack) = top(X - 1, pop(Stack))} \qquad \text{[top-n]}$$

$$\frac{X < 0 = 1}{top(X, Stack) = 0} \qquad \text{[top-error]}$$

$$pop(push(X, Stack)) = Stack \qquad \text{[pop-0]}$$

$$pop(empty) = empty \qquad \text{[pop-error]}$$

$$\frac{X < 1 = 0}{pop(X, Stack) = pop(X - 1, pop(Stack))} \qquad \text{[pop-n]}$$

$$\frac{X < 1 = 1}{pop(X, Stack) = Stack} \qquad \text{[pop-1-error]}$$

$$drop(X,\, Stack) = push(top(Stack),\, pop(X+1,\, Stack)) \qquad \text{[drop-0]}$$

$$\frac{\begin{array}{c} X < 1 = 0, \\ take(X-1,\, pop(Stack)) = \{Edge^*\} \end{array}}{take(X,\, Stack) = \{Edge^*,\, X \rightarrow top(Stack)\}} \qquad \text{[take-1]}$$

$$take(X,\, Stack) = \{\} \qquad \textbf{otherwise} \qquad \text{[take]}$$

Apart from the operations *drop* and *take*, and the use of the index 0 to reference the top, this stack specification is completely standard.

Stacks are built from the constructor functions *empty* and *push*. The stack is either empty, or it contains references (by number) to nodes in the graph. If a number on the stack is larger than the number of any node in the graph, it is taken to be a forward reference.

The functions *top* and *pop* are defined in a fairly usual way. However, the functions *drop* and *take* are unusual. The function *drop* pops elements off the stack, but preserves the topmost element. The function *take* produces a list of edges from a stack and a SIG.

7. GEL syntax and semantics

Now we will discuss GEL command by command. We will give the semantics of GEL by specifying how the state of the abstract machine defined in Section 6 is affected by reading a single line. Given the specification of the abstract machine, we need only one equation for the specification of the semantics of one GEL command.

7.1 An overview of GEL commands

We will present the overview of GEL commands by annotating separate context-free syntax sections in the module Gel.

Module Gel
imports Gel-types(5.1)
exports
sorts GEL-ITEM GEL
context-free syntax
GEL-ITEM* → GEL

A full GEL text is a sequence of GEL-ITEMs. Every GEL-ITEM is terminated by a carriage return. Because of lexical syntax problems (see Appendix C) this return is put in a sort RET, which is fully defined only in module Full-Gel-syntax. In all other modules, only variables and meta-variables of this sort can be used.

Comments

context-free syntax

"%" TYPE RET → GEL-ITEM

Comments are ignored. The library in Appendix B inserts a comment with version information.

The abbreviation command

context-free syntax

"!" SHORT-TYPE ":" INT "=" TYPE RET → GEL-ITEM

Primarily, the compactness of GEL is achieved by its abbreviation mechanism. For every combination of type and arity occurring in a graph described by a GEL text, an abbreviation is introduced by the abbreviation command. The syntax for the abbreviation command is introduced in the context-free syntax section above. The effect of this command is defined in Section 7.3.

To GEL, the type of a node is uninterpreted, but for the graph structure, it *is* important how many edges depart from a node. Somewhat redundantly, this is specified both by an arity (sort INT) in the definition of an abbreviation and by the actual edges departing from the node. Positive values denote fixed arities, -1 denotes varyadic arity (with an alias *), and all other values are used as error values. In Section 11, we will discuss the concerns leading to this redundant specification.

The build commands

context-free syntax

SHORT-TYPE RET → GEL-ITEM
SHORT-TYPE SPACE INT RET → GEL-ITEM

There are two versions of the build command, one for fixed and one for varyadic arities. The varyadic variant has an integer argument specifying the actual number of edges to make. The effect of these commands is defined in Section 7.4.

The copy commands

context-free syntax

"#" INT RET → GEL-ITEM
">" INT RET → GEL-ITEM

For the expression of sharing and circularities, there are two copy commands, one for backward references (introduced by '#') and one for forward references (introduced by '>'). The parameter of a backward reference denotes an offset in the stack, where the reference to be copied can be found. The parameter of a forward reference denotes the number of build commands to be processed until the actual node will be produced.

The drop command

context-free syntax

"*" INT RET → GEL-ITEM

When nodes with non-zero arities are built, references are popped off the stack. It is not always possible to put the references on the stack in such a way that only one reference remains on the stack after the last build command. To this purpose, the drop command removes a number of references just below the top reference on the stack.

7.2 GEL semantics

Module Gel-read
imports Gel$^{(7.1)}$ Graphs$^{(5.2)}$ Stacks$^{(6.2)}$ Tables$^{(6.1)}$
exports
context-free syntax
read GEL → GRAPH

hiddens
context-free syntax
read GEL *with next* INT "," *stack* STACK "," *abbreviations* TABLE *and nodes* NODES → GRAPH
variables
Gel-Item [*0-9′*]*"*"* → GEL-ITEM*
Gel [*0-9′*]* → GEL
equations

read Gel = *read Gel with next* 1, *stack empty*, *abbreviations empty-table and nodes* [] [read-0]

read with next X, *stack push*(*X′*, *empty*), *abbreviations Table and nodes* [*Node**] = [extract-0]
root is X′ in [*Node**]

read Gel with next X, *stack Stack*, *abbreviations Table and nodes* [*Node**] = *error_graph* [extract-1] **otherwise**

As a hidden function, the module Gel-read contains the constructor function for a state of the GEL machine. The first argument is the GEL-text still to be read, the second argument is the number of the next node to build, the third argument a stack of references to subgraphs, the fourth argument a table of abbreviations for types and signatures, and the fifth arguments contains the nodes that have been built until now. Final states have exactly one reference on the stack, and an empty GEL text, otherwise the input GEL text was erroneous. In the following subsections, we will use a running example to illustrate the effects of the commands. The initial state of the GEL machine reading this example is given in Figure 5.

7.3 Effects of the abbreviation command
Abbreviations are handled by the equation

read ! *Short-Type* : X_1 = *Type Ret*
*Gel-Item**
with next X, *stack Stack*, *abbreviations Table and nodes* [*Node**] = [read-abbr]
*read Gel-Item** *with next X*, *stack Stack*,
abbreviations define Short-Type of type Type with sig X_1 *in Table*
and nodes [*Node**]

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS

read a <RET>
b <SPACE> 0 <RET>
c <RET>
0 <RET>
> 1 <RET>
c <RET>
1 <RET>
b <SPACE> 2 <RET>
∗ 1 <RET>
with next 1,
stack empty,
abbreviations [*abbr* c *of type* PLUS *with sig* 2,
abbr b *of type* LIST *with sig* ∗,
abbr a *of type* ONE *with sig* 0]
and nodes []

Figure 6: The GEL machine after reading 3 abbreviations

In this equation, only the abbreviation table is updated. Thus, after reading abbreviations for a type ONE with arity 0, a type LIST of varyadic arity, and a type PLUS with named edges left and right, the state of the GEL machine is as shown in Figure 6.

read ! a : 0 = ONE <RET>
! b : ∗ = LIST <RET>
! c : 2 = PLUS <RET>
a <RET>
b <SPACE> 0 <RET>
c <RET>
0 <RET>
> 1 <RET>
c <RET>
1 <RET>
b <SPACE> 2 <RET>
∗ 1 <RET>
with next 1,
stack empty,
abbreviations []
and nodes []

Figure 5 An initial state of the GEL reader

7.4 Effects of the build commands

Build commands are handled by the equations

$$\frac{X = \textit{sig of Short-Type in Table},\quad X < 0 = 0}{\begin{array}{l}\textit{read Short-Type Ret}\\ \quad \textit{Gel-Item}^*\\ \textit{with next } X_1,\ \textit{stack Stack},\ \textit{abbreviations Table and nodes } [\textit{Node}^*] =\\ \textit{read Gel-Item}^*\\ \textit{with next } X_1 + 1,\\ \quad \textit{stack } \textit{push}(X_1,\ \textit{pop}(X,\ \textit{Stack})),\\ \quad\quad \textit{abbreviations Table}\\ \textit{and nodes } [\textit{Node}^*,\ \textit{node } X_1 \textit{ of type type of Short-Type in Table}\\ \quad\quad \textit{with sig } X\\ \quad\quad \textit{and edges } \textit{take}(X,\ \textit{Stack})]\end{array}} \quad \text{[read-bld-fix]}$$

$$\frac{\textit{sig of Short-Type in Table} < 0 = 1}{\begin{array}{l}\textit{read Short-Type Ret}\\ \quad \textit{Gel-Item}^*\\ \textit{with next } X_1,\ \textit{stack Stack},\ \textit{abbreviations Table and nodes } [\textit{Node}^*] =\\ \textit{error\_graph}\end{array}} \quad \text{[read-bld-fix-error]}$$

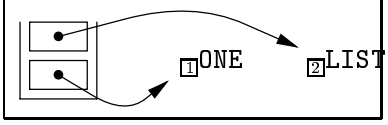
$$\frac{\textit{sig of Short-Type in Table} = -1}{\begin{array}{l}\textit{read Short-Type Space } X \textit{ Ret}\\ \quad \textit{Gel-Item}^*\\ \textit{with next } X_1,\ \textit{stack Stack},\ \textit{abbreviations Table and nodes } [\textit{Node}^*] =\\ \textit{read Gel-Item}^*\\ \textit{with next } X_1 + 1,\\ \quad \textit{stack } \textit{push}(X_1,\ \textit{pop}(X,\ \textit{Stack})),\\ \quad\quad \textit{abbreviations Table}\\ \textit{and nodes } [\textit{Node}^*,\ \textit{node } X_1 \textit{ of type type of Short-Type in Table}\\ \quad\quad \textit{with sig sig of Short-Type in Table}\\ \quad\quad \textit{and edges } \textit{take}(X,\ \textit{Stack})]\end{array}} \quad \text{[read-bld-var]}$$

$$\frac{\textit{sig of Short-Type in Table} \neq -1}{\begin{array}{l}\textit{read Short-Type Space } X \textit{ Ret}\\ \quad \textit{Gel-Item}^*\\ \textit{with next } X_1,\ \textit{stack Stack},\ \textit{abbreviations Table and nodes } [\textit{Node}^*] =\\ \textit{error\_graph}\end{array}} \quad \text{[read-bld-var-error]}$$

The equations describing the effect on the GEL machine are very similar. For types of fixed arity (equation `read-bld-fix`), the number of items to be popped of the stack is looked up in the abbreviation table, whereas for varyadic types (equation `read-bld-var`), this number is taken from the command. Similarly, the edge names are taken from the signature for types of fixed arity, whereas for varyadic types the edge names are numbered according to the command. Because the build command actually builds a new node, the count of nodes is increased by one in both equations.

In our example, the reading of two build commands containing abbreviations of a fixed and a varyadic type results in the state shown in Figure 7.

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS

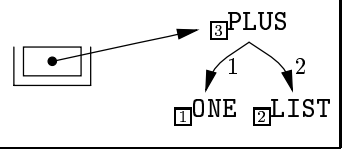


read c <RET>
  # 0 <RET>
  > 1 <RET>
  c <RET>
  # 1 <RET>
  b <SPACE> 2 <RET>
  * 1 <RET>
with next 3,
  stack push(2, *push*(1, *empty*)),
  abbreviations [*abbr* c *of type* PLUS *with sig* 2,
    abbr b *of type* LIST *with sig* *,
    abbr a *of type* ONE *with sig* 0]
and nodes [*node* 1 *of type* ONE *with sig* 0 *and edges* {},
  node 2 *of type* LIST *with sig* * *and edges* {}]

Figure 7 The GEL machine after 3 abbreviations and 2 builds

Note that after each build command, a reference to the node built is left on the stack. These references are popped on build commands for non-zero arities, e.g. reading of a c on the next line of our example results in the state displayed in Figure 8.

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS



read # 0 <RET>
  > 1 <RET>
  c <RET>
  # 1 <RET>
  b <SPACE> 2 <RET>
  * 1 <RET> *with next* 4,
    stack push(3, *empty*),
    abbreviations [*abbr* c *of type* PLUS *with sig* 2,
      abbr b *of type* LIST *with sig* *,
      abbr a *of type* ONE *with sig* 0]
and nodes [*node* 1 *of type* ONE *with sig* 0 *and edges* {},
  node 2 *of type* LIST *with sig* * *and edges* {},
  node 3 *of type* PLUS *with sig* 2
  and edges {1 → 1,
    2 → 2}]

Figure 8 The GEL machine after 3 abbreviations and 3 builds

7.5 Effects of the copy commands

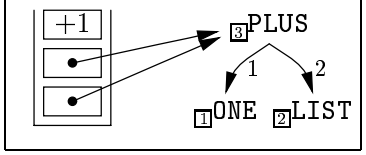
The meaning of the copy commands is specified in the equations

read # *X Ret*
  Gel-Item*
with next X_1, *stack Stack*, *abbreviations Table and nodes* [*Node**] = [read-scopy]
  read Gel-Item* *with next* X_1, *stack push*(*top*(*X*, *Stack*), *Stack*), *abbreviations Table and nodes* [*Node**]

read > *X Ret*
  Gel-Item*
with next X_1, *stack Stack*, *abbreviations Table and nodes* [*Node**] = [read-fcopy]
  read Gel-Item* *with next* X_1, *stack push*(X_1 + *X*, *Stack*), *abbreviations Table and nodes* [*Node**]

In ASF+SDF terms, the subsequent reading of a forward and a backward reference leads to the state shown in Figure 9

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS

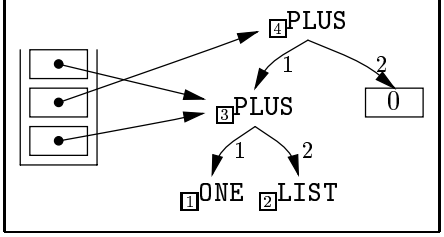


read `c` <RET>
  # 1 <RET>
  `b` <SPACE> 2 <RET>
  ∗ 1 <RET>
with next 4,
  stack push(5, *push*(3, *push*(3, *empty*))),
    abbreviations [*abbr* `c` *of type* `PLUS` *with sig* 2,
      abbr `b` *of type* `LIST` *with sig* ∗,
      abbr `a` *of type* `ONE` *with sig* 0]
and nodes [*node* 1 *of type* `ONE` *with sig* 0 *and edges* {},
  node 2 *of type* `LIST` *with sig* ∗ *and edges* {},
  node 3 *of type* `PLUS` *with sig* 2
  and edges {1 → 1,
    2 → 2}]

Figure 9 The GEL machine after 3 abbreviations, 3 builds and 2 copies

Forward references are depicted as rectangular boxes containing the number of nodes to be built before the actual node will be built. Therefore, reading two more lines, we get the state displayed in figure 10.

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS



read `b` <SPACE> 2 <RET>
  ∗ 1 <RET>
with next 5,
  stack push(3, *push*(4, *push*(3, *empty*))),
    abbreviations [*abbr* `c` *of type* `PLUS` *with sig* 2,
      abbr `b` *of type* `LIST` *with sig* ∗,
      abbr `a` *of type* `ONE` *with sig* 0]
and nodes [*node* 1 *of type* `ONE` *with sig* 0 *and edges* {},
  node 2 *of type* `LIST` *with sig* ∗ *and edges* {},
  node 3 *of type* `PLUS` *with sig* 2
  and edges {1 → 1,
    2 → 2},
  node 4 *of type* `PLUS` *with sig* 2
  and edges {1 → 3,
    2 → 5}]

Figure 10 The GEL machine after 3 abbreviations, 3 builds and 2 copies, a build and one more copy

Note, that the future has come one build-step closer; the `right` edge of the `PLUS` node now refers to the next node that will be built.

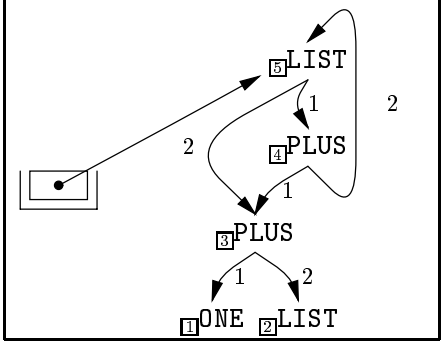
7.6 Effects of the drop command

The meaning of the drop command is specified in the equation

read ∗ *X Ret*
  Gel-Item*
with next X_1, *stack Stack*, *abbreviations Table and nodes* [*Node**] = [read-drop]
read Gel-Item* *with next* X_1, *stack drop*(*X*, *Stack*), *abbreviations Table and nodes* [*Node**]

The result of reading the last two lines in the example is shown in figure 11. Note that the GEL text is erroneous if more than one element is left on the stack after the text is processed.

SHORT-TYPE	SIG	TYPE
a	0	ONE
b	-1	LIST
c	2	PLUS




```
read
with next 6,
        stack push(5, empty),
            abbreviations [abbr c of type PLUS with sig 2,
                           abbr b of type LIST with sig *,
                           abbr a of type ONE with sig 0]
and nodes [node 1 of type ONE with sig 0 and edges {},
           node 2 of type LIST with sig * and edges {},
           node 3 of type PLUS with sig 2
           and edges {1 → 1,
                      2 → 2},
           node 4 of type PLUS with sig 2
           and edges {1 → 3,
                      2 → 5},
           node 5 of type LIST with sig *
           and edges {1 → 4,
                      2 → 3}]
```

Figure 11: The GEL machine after 3 abbreviations, 3 builds and 2 copies, a build, one more copy and a drop

8. Evaluation of the specification

We would like to note the following about the specification of GEL reading above.

- The specification should not leave any doubts concerning the mapping of a GEL text into a graph.
- The mapping is overspecified. Firstly, any consistent renumbering of nodes should be considered the same graph. Secondly, there is no need for an implementation to keep track of subgraphs after their root disappears from the stack, because they cannot be referenced by any GEL command. In the C implementation described in the appendix, no explicit graph is maintained. Instead, some bookkeeping is done only for forward references.
- The lexical syntax of GEL could not be specified satisfyingly. It is impossible to have carriage returns occur both in the definition of LAYOUT and in the meaningful part of the syntax.

9. A GEL writing algorithm

In the previous sections, we have presented a mapping from GEL texts to graphs (the function *read*). This mapping is an exact definition of the meaning of GEL texts. In general, many GEL texts are mapped (by *read*) to the same graph. Therefore, there is no unique inverse function of *read*. However, given a graph, there is always at least one GEL text that describes it. We will not prove this here.

Here we give the pseudo code implementing a function *write* that satisfies the requirement that for all graphs g, g=`read`(`write`(g)). The pseudo code is equivalent to the algorithm implemented in the library described in Appendix B. Apart from the correctness requirement above, this algorithm satisfies the following requirements:

- Forward references are only generated for edges pointing to ancestors in the depth-first tree associated with the graph (see [CLR89] for this terminology). Given the fixed order in which children are visited, this is a minimal number of forward references.
- Nodes with a single parent are written in the order they are needed by their parent, no copy commands are generated for them.

For trees, this algorithm uses no more stack space than the total number of nodes. For balanced trees, it uses only logarithmic stack space. If less than 250 different types occur in an instance, the length of a GEL text asymptotically approximates one character per node when the number of nodes becomes large, and the number of edges is comparable to the number of nodes. The algorithm is divided in three phases:

Phase1 A depth-first traversal is done to detect which nodes are shared. Every node is ordered into a sequence after all its edges have been explored.

Phase2 For every shared node, it is determined how many 'local' nodes are visited in a depth-first traversal starting at the shared node, avoiding subgraphs accessible through shared nodes.

Phase3 The subgraphs associated with the shared nodes are written in the sequence obtained in Phase1. Stack-offsets for shared nodes that have already been written are determined by a simple model of the stack-layout of the GEL-reader, offsets for shared nodes that will be written in the future are determined by the number of intervening 'local' nodes (see Phase2). Finally, if the root node is not a shared node, the subgraph associated with the root node is written.

In pseudo code, the algorithm reads as follows. Global variables are *shared, entered, sequence, onstack, write_seq, written* and *abbreviated*. Procedure Phase1 is called on the root of the graph, procedure Phase2 and Phase3 are called without arguments. In the pseudo code, "*string1*<*exp*>*string2*" denotes the string obtained by concatenating *string1*, the (string) value of *exp*, and *string2*.

```
shared = {}          % set of shared nodes
entered = {}         % set of entered nodes
sequence = []        % visit order of nodes
onstack = 0          % number of refs on reader stack
next_build = 1       % number of next build
written = {}         % set of nodes already written
abbreviated = {}     % set of types with abbreviation

extern varyadic(n)   % test if node n is varyadic
extern sig(t)        % signature of type t
extern abbr(t)       % unique abbreviation for type t
extern type(n)       % type of node n
extern size(l)       % number of elements in set or list
extern children(n)   % list of edges emanating from node n

phase1(n) {
   if ((n ϵ entered) and not (n ϵ shared))
   then shared = shared ∪ {n}
   else
      entered = entered ∪ {n}
      for r in children(n): phase1(r)
      sequence = sequence :: n
```

```
    fi
}

int locals(n) { % auxiliary for phase 2
    childrensum = 0
    for r in children(n):
        if (r ∉ shared)
        then childrensum = childrensum + locals(r)
        fi
    return childrensum + 1
}

phase2() {
    cumlocs = 0
    sharedno = 1
    for r in sequence
        if (r ∉ shared)
            cumlocs = cumlocs + locals(r)
            r.cum = cumlocs
            r.sharedno = sharedno
            sharedno = sharedno + 1
        fi
}

write(n) { % auxiliary for phase3
    for r in children(n)
        if (r ∈ shared)
        then if (r ∈ written)
                then print "# <onstack-n.sharedno> \n"
                else print "> <n.cum-next_build> \n"
                fi
            onstack = onstack + 1
        else write(r)
        fi
    if (type(n) ∉ abbreviated)
    then
        print "!<abbr(type(n))>:<sig(type(n))>=<type(n)> \n"
        abbreviated = abbreviated ∪ {type(n)}
    fi
    print "<abbr(type(n))>"
    if varyadic(n) then print "␣ <size(n)> \n" else print "\n" fi
    next_build = next_build + 1
    onstack = onstack - (size(n)-1)
}

phase3() {
for r in shared write(r)
if root ∉ shared
then
    write(root)
    print "* <number(shared)-1> \n"
else
    print "* <number(shared)\n"
fi
}
```

10. GEL writing modulo unraveling

In the preceding section, we have shown a correct `write` function, yielding the identity when composed with the `read` function. In some cases, however, it is sufficient if the composition of

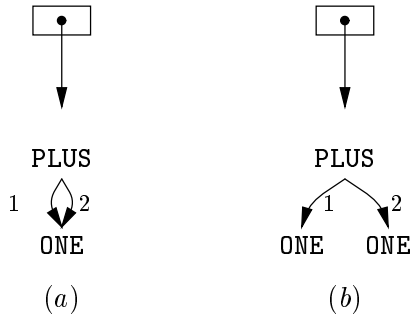


Figure 12: Two graphs with the same unravelling

`read` and `write` yields a graph in the same equivalence class as the input, given an equivalence relation on the graphs.

Examples are term and graph rewriting, where (term) graphs are equivalent if they are in the rewrite relation. Therefore, in a graph rewriting implementation that uses GEL, there is no need to build or write the graph as specified in the `read` specification, any equivalent graph (e.g. the normal form) will do.

Also, several graphs may represent the same term (sharing). Two graphs that are nontrivially equivalent in this sense are shown in Figure 12.
In graph (a), the node with type `ONE` is shared by two edges, whereas in graph (b), both edges lead to a private copy. Depending on the context, both the introduction of sharing and the removal of sharing can be advantageous to the size of the GEL text. Below, the GEL texts for graphs (a) and (b) are shown:

```
! a : 0 = ONE <RET>      ! a : 0 = ONE <RET>
a <RET>                  a <RET>
# 0 <RET>                a <RET>
! b : 2 = PLUS <RET>     ! b : 2 = PLUS <RET>
b <RET>                  b <RET>
```

In this example, the expression of the sharing of a single node takes more space than simply writing out two copies (line 3 of the left GEL text takes at least two bytes in the binary encoding, whereas line 3 of the right text takes only one byte). When larger graphs are shared, the reverse becomes true.

Formally, two graphs g_1 and g_2 represent the same term (up to isomorphism) if they have the same *unraveling*. Following [KKSdV93], we first define paths

> A path in a graph g is a finite or infinite sequence $a, i, b, j, \ldots$ of alternating nodes and integers, beginning and (if finite) ending with a node of g, such that for each m, i, n in the sequence, where m and n are nodes, n is the ith successor of m. If the path starts from a node m and ends at a node n, it is said to be a path from m to n.

Given the notion of paths, the *unraveling* $U(g)$ of a graph g is defined as:

> The unraveling $U(g)$ of a graph g is the term representation of the following forest. The nodes of $U(g)$ are the paths of g which start from the root. Given a node

$a, i, b, j, \ldots, y$ of $U(g)$, then this node has the same type as y, and its successors are all paths of the form $a, i, b, j, \ldots, y, n, z$, where z is the nth successor of y in g.

This notion is defined without reference to peculiarities of a particular application of GEL, and therefore, the library implementation of GEL can exploit it. This does not imply that graphs with the same unraveling are considered equivalent in all applications of GEL, and therefore writing modulo unraveling is parameterized by two predicates on nodes, `unshare` and `compress`. If both predicates are false on all nodes, `read` $\circ$ `write` is the identity. If `unshare` is true of some node, multiple edges to a subgraph are unraveled by `read` $\circ$ `write` into edges to graphs with the same unraveling. If `compress` is true of some node, multiple edges to a subgraph in the image correspond to edges to some subgraphs with the same unraveling in the origin. In a table:

$\exists i : \texttt{unshare}(i)$	$\forall i : \texttt{not}(\texttt{unshare}(i) \vee \texttt{compress}(i))$	$\exists i : \texttt{compress}(i)$
$i_1 = i_2 \Rightarrow o_1 \simeq o_2$	$i_1 = i_2 \Leftrightarrow o_1 = o_2$	$i_1 \simeq i_2 \Leftarrow o_1 = o_2$

Where a subscripted i refers to a subgraph in the input, a subscripted o refers to the corresponding subgraph in the output, and $g_1 \simeq g_2$ means that g_1 and g_2 have the same unraveling. Note that the use of unraveling equivalence is enabled, but not forced by the parameters. This is because determining if graphs have the same unraveling may be far too expensive.

11. Discussion of design decisions

At first sight, there seems no need for the *definition* of an abbreviation to occur in a GEL text. Processing the definition of an abbreviation takes time, so one might argue that if the reader and writer agree in advance on the full names of a `TYPE`, they might as well agree on their abbreviations.

However, there are four good reasons for GEL's abbreviation scheme:

1. A scheme with fixed abbreviations deteriorates when there are many readers and writers using different sets of types. Then, a globally consistent set of abbreviations must be found, resulting in longer abbreviations, and awkward recompilations if new components are added to the system.

2. In GEL's scheme, the number of bits occupied by an abbreviation is determined by the number of `TYPE`s actually occurring in a single graph. This is typically one or two orders of magnitude smaller than the total number of `TYPE`s known in the system. If the number of actually occurring `TYPE`s is still large, the redefinability of abbreviations may be used to keep the `SHORT-TYPE`s small.

3. The abbreviation mechanism is convenient for the exchange of 'external' types like bitmapped pictures.

4. GEL texts are completely self-describing, enabling inspection with tools that are independent of the actual application.

It should be noted that GEL's abbreviation scheme deteriorates to a worst case when every node in the graph has a unique `TYPE`. In the systems we know, this happens only in the case of very small graphs (such as the examples given at the start of this paper), where efficiency is not a severe problem. Even in this worst case, the overhead is only 4 bytes per node.

From the viewpoint of efficiency, one might also argue that the arity should not be specified, because it could be a function of the `TYPE`, known to both reader and writer. However, the specification of arities allows decoding of GEL texts without interpreting `TYPE`s. It should be noted that the arity is specified only once for every type, thus causing a relatively small loss in efficiency.

From a minimalistic point of view, only varyadic arities are needed, because any graph structure can be specified with varyadic types. The introduction of fixed arities, however, allows shorter encodings because the actual number of edges need not be specified for every node.

The copy commands use indices relative to the stack (for backward references) and relative to the building sequence (for forward references). In the writer, this is more difficult to implement than a label scheme as used in, e.g., the ERL of IDL. However,

- Labels have definition and use occurrences, whereas relative indices have only use occurrences. Thus, relative indices yield a shorter representation.
- Stack references by number are cheaper to implement in the reader than label references, forward references are probably equally expensive as label references.
- Relative indices permit compositionality of term graphs; for a term C[t1,t2], the GEL text is the concatenation of the GEL texts for t1, t2 and C. No relabelings are needed. It might be interesting to investigate the support of more powerful graph compositions.
- Even if writing is more expensive, it is to be expected that a GEL representation will be read at least as many times as it is written.

12. Measurements

For GEL texts describing graphs of several sizes, we have made measurements of reading and writing time (sum of system time and cpu time), and the number of bytes per node (in the binary representation). The measurements were performed on a Sillicon Graphics Indigo, with a MIPS R4000 processor running at 100 Mhz internal clock frequency. We used the C library presented in Appendix B for the measurements of reading and writing times.

For the sake of comparison, in the last row of the tables, figures are given for a fast, application specific binary writer and reader of binary trees. These programs use fixed codes for the set of types occurring in their trees, and read or print the codes as they are encountered in a preorder traversal. This format is widely known as 'Polish' notation, therefore we have called the programs 'Polish'. Finally, in the last column of the tables, we show the ratio between the GEL functions and the 'Polish' programs.

# nodes	bytes	reading time	ratio
23	17	60.5 ± 0.5	19.8
4471	1.49	6.76 ± 0.35	2.21
68947	1.24	6.75 ± 0.04	2.21
Polish (4095)	1	3.06 ± 0.01	1.00

Table 1. GEL reading time (in μs) per node

# nodes	write (cpu+sys)	ratio
23	723 ± 25	520
4471	14.2 ± 0.8	10.1
68947	17.9 ± 1.0	12.7
Polish (4095)	1.41 ± 0.04	1.00

Table 2. GEL writing time (in μs) per node

In the second column of table 1, we see that the number of bytes per node is high (17) for small graphs, but tends to 1 byte per node for large graphs. Related to this, the processing time per node is much higher for small graphs than for large graphs. We observe that writing is between 2 and 3 times as expensive as reading. Only a small part of the difference can be attributed to the fact that writing a file is more expensive than reading it. It seems more likely that the sharing analysis is expensive. Some more tuning and profiling might reduce the gap between reading and writing.

It is interesting to compare these figures to the figures found by Lamb in his Ph.D. thesis [Lam83], for a signature that is comparable to ours. We have multiplied Lamb's number of bytes per node with 7/8, because Lamb used a 7-bit machine (running at about 2 MIPS). Lamb tested an ASCII writer, a binary writer (binary encodings are given for the node types), a hand-coded dedicated Polish tree-writer (in a pre-order traversal, only a code for the node type is written), and the LG package of the PQCC project [N$^+$78]. The sizes of his structures were between 400 nodes and 10000 nodes, and the values reported are the result of applying linear regression techniques. This implies that Lamb's software is not very sensitive to the size of the graphs.

Package	bytes/node	Input (ms)	ratio
ASCII	14.72	2.38 ± 0.003	21
Binary	12.18	$0.981 \pm .002$	8.8
Polish	2.61	0.111 ± 0.0002	1.0
PQCC	25.43	2.028 ± 0.007	18

Table 3. Lamb's readers

For larger graphs, in terms of the number of bytes per node, our GEL implementation still scores about twice as good as Lamb's Polish writer. This can only be explained if Lamb's codes for the node types are 2.61 bytes long on average. Even for small graphs, GEL does not perform significantly worse than any of the other three packages.

Package	Output (ms)	ratio
ASCII	1.434 ± 0.002	28
Binary	0.360 ± 0.002	7.1
Polish	0.051 ± 0.0003	1.0
PQCC	1.322 ± 0.010	26

Table 4. Lamb's writers

With regard to the processing times, we will only make relative comparisons. That is, we will compare the ratios to the 'polish' readers and writers. First we will consider 'average' terms, of about 4000 nodes. Lamb's ASCII reader performs 21.4 times as bad as his polish reader, whereas our GEL reader performs only 2.2 times as bad as our polish reader. Lamb's ASCII writer performs 28.1 times as bad as his polish writer, whereas our GEL writer performs only

10 times as bad as our polish writer. We attribute this to the fact that the GEL formalism is much closer to polish notation than ERL.

We draw the conclusion that it is hardly ever worthwhile to hand-code a GEL reader for a production version, but hand-coding a GEL writer can be useful. Hand-coding has the additional advantage that maximal knowledge about sharing can be exploited. Of course, if the internal graph representation is such that almost uninterpreted memory dumps can be made, and both reader and writer use the same representation, techniques as described in [New87] can be used.

13. Conclusions

GEL is a formalism for the implementation-language independent exchange of graph-structured data. GEL is exclusively concerned with graph-structure, the type of a node is a sequence of bytes, uninterpreted by GEL. The formal semantics of GEL allows an easily verifiable implementation of GEL readers and writers.

GEL is compositional. Especially in generated distributed environments, this is important. There, it often happens that input graphs must be composed of several output graphs.

Asymptotically, GEL representations of large, treelike graphs tend to require only one byte storage for representing one node in the graph.

The speed and compactness of GEL should, in almost all cases, overcome the need for alternative implementations for production versions of tools.

Special thanks go to Job Ganzevoort, Steven Klusener, Paul Klint and Pum Walters, for numerous suggestions and discussions. Job implemented the initial version of the `C` library, Steven turned the formal specification of GEL inside out. Remaining errors are of course the responsibility of the author.

References

[BHK89a] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BHK89b] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[BK94] J.A. Bergstra and P. Klint. The toolbus - a component interconnection architecture. Technical Report P9408, Universiteit van Amsterdam, P.O. Box 41882, 1009 DB Amsterdam, The Netherlands, March 1994.

[BvEJ+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W.Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In J.W. de Bakker, A.J. Nijman, and vol. II P.C. Treleaven, editors, *Proceedings PARLE'87 Conference*, number 259 in Lecture Notes in Computer Science, pages 141–158, Eindhoven, 1987. Springer Verlag.

[CCI] CCITT. Specification of basic encoding rules for abstract syntax notation one (asn.1). Recommendation X.209, technically aligned with ISO 8825.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT press, Cambridge Massachussetts, 1989.

[DR94] A.M. Dery and L. Rideau. A message protocol for distributed programming environments. Technical report, GIPE II, Generation of Interactive Programming Environments, phase 2, sixth review report, january 1994. ESPRIT project 2177.

[Hen89] P.R.H. Hendriks. Lists and associative functions in algebraic specifications - semantics and implementation. Report CS-R8908, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1989.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[KKSdV93] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. The adequacy of term graph rewriting for simulating term rewriting. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*. John Wiley & Sons Ltd, 1993.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[Lam83] D.A. Lamb. *Sharing Intermediate Representations: The Interface Description Language*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1983.

[N+78] J.M. Newcomer et al. PQCC implementor's handbook. Technical report, CMU, 1978. Internal Report.

[New87] Joseph M. Newcomer. Efficient binary i/o of IDL objects. *SIGPLAN Notices*,

22(11):35–43, nov 1987.

[Sno89] Richard Snodgrass. *The Interface Description Language, Definition and Use.* Principles of Computer Science Series. Computer Science Press, Rockville, MD 20850, 1989.

[Wal94] H.R. Walters. A complete term rewriting system for decimal integer arithmetic. Technical Report CS R9435, CWI, 1994.

A. Binary encoding of GEL

For a realistic implementation, a binary encoding of GEL was designed. In the binary encoding, numeric (binary) abbreviations are used instead of textual abbreviations, and a fixed format caters for fast reading.

For conversion to and from the textual format, there are tools `gel2a` (GEL to ascii) and `gel2b` (GEL to binary). Both tools have an option `-b` for an even older variant of the textual format that uses binary naturals in ascii ("0", "1", "10" etc.).

We will use the following notations in the definition of the binary format:

Notation	Meaning
`<n>`	A byte with value `n`.
`<'c'>`	A byte with the ASCII-value of the character `c`.
`xx<n>`	A byte starting (most significant bits) with `xx`, followed by a binary value `n` (remaining bits in the byte).
`#-k-`	A sequence of one or more bytes defining a positive number or index k, defined as follows: `0<n>` where $k = n$ and $(0 \leq k \leq 127)$ `10<n1> <n2>` where $k = 256 * n1 + n2$ and $(0 \leq k \leq 16383)$ `110<n1> <n2> <n3>` where $k = 256 * (256 * n1 + n2) + n3$ and $(0 \leq k \leq 2M)$ `1110<n1> <n2> <n3> <n4>` where $k = 256 * (256 * (256 * n1 + n2) + n3) + n4$ and $(0 \leq k \leq 256M)$ etc. `11111110<n1> .. <n7>` for $(0 \leq k <= 2^{56} - 1)$
$$$	A byte-sequence `c1..ck`, encoded as: `#-k- <c1> <c2> ... <ck>` where `k` is the length of the sequence, followed by the sequence itself.

Given this notation, we shall now describe the binary format of GEL texts. A binary GEL text is an unseparated sequence of GEL items, where every item is either the introduction of an abbreviation, some stack operation, or a build command.

In the important case that the graphs are trees, the build command occurs most often, so it is important that the encoding of a build command is as short as possible. Build commands are encoded by a single number for the abbreviation (and an additional number for the arity in case of varyadic types), with the provision that the numbers $0 \ldots 10$ are reserved as tags for the other, less frequently occurring commands.

A.1 Build commands

Encoding	Meaning
`#-k-`	$(10 < k < 2^{56} - 1)$, build node given the abbreviation no. `k` of a fixed arity type.
`#-k- #-l-`	$(10 < k < 2^{56} - 1)$, build node given the abbreviation no `k` of a varyadic arity type, with `l` children.

A.2 Abbreviations

Encoding	Meaning
`<0> #-s- #-k- $$$`	Introduction of abbreviation no. `s (s>10)` for fixed-arity type with `TYPE` $$$ and arity `k`
`<1> #-s- $$$`	Introduction of abbreviation no. `s (s>10)` with variadic arity and type `$$$`

A.3 Stack operations

Encoding	Meaning
`<2> #-k-`	duplicate stackitem at index `k` (top of stack is 0).
`<3> #-k-`	forward reference, index `k` (next node built is 0).
`<4> #-k-`	drop k items from stack.

A.4 Comments

Encoding	Meaning
`<5> $$$`	the comment `$$$`.

A.5 An example of a binary encoding

As an example, Figure 13 on page 31 displays the binary encoding of the GEL text that was used as the running example in this document.

Binary Code	Description
`<0><11><0>`	Abbreviation 6 with sig 0
`<3><'O'><'N'><'E'>`	of type `ONE`.
`<1><12>`	Abbreviation 7 with sig *
`<4><'L'><'I'><'S'><'T'>`	of type `LIST`.
`<0><13><2>`	Abbreviation 8 with sig 2
`<4><'P'><'L'><'U'><'S'>`	of type `PLUS`.
`<11>`	build a node of type `ONE`
`<12><0>`	build a node of type `LIST` with 0 edges
`<13>`	build a node of type `PLUS`
`<2><0>`	reference top of stack
`<3><1>`	forward reference
`<13>`	build another `PLUS`
`<2><1>`	reference in stack
`<12><2>`	build a node of type `LIST` with 2 edges
`<5><1>`	drop one element from the stack

Figure 13: A binary encoding of the running example

B. Using the C implementation of GEL: <gel.h>

We have implemented a C library of re-entrant functions for reading and writing of GEL representations; multiple graphs can be written concurrently. In most cases, this library will overcome the need to write an application specific reader or writer for GEL. The application specific part that needs to be written is typically a few pages of C.

To allow arbitrary internal representations of graphs for every application using the GEL library, the library does not maintain a complete representation of the graph, nor a complete representation of the abbreviation table. Instead, this is left to the application using the library.

For references to subgraphs of the graph maintained by the application, the GEL library uses a `C` type `gel_node`. During reading, when a new node must be built by the application, the edges are supplied by the library as an array of values of the `gel_node` type, and a reference to the new node is returned by the application as a new value of this type. Forward references are handled by supplying a new reference for a certain edge of an existing node. During writing, the GEL library traverses the application's graph by calling functions defined on values of the `gel_node` type.

For the implementation of the abbreviation table, the GEL library maintains a correspondence between `SHORT-TYPE`s and values of a C type `gel_type`, and the application maintains a correspondence between values of the type `gel_type` and corresponding `TYPE`s (sequences of bytes) and `SIG`s (integers, or varyadic). On reading an abbreviation, the GEL library asks the application to supply a `gel_type` for a combination of a `TYPE` and a `SIG`. This `gel_type` is passed to the application when a new node must be built.

During writing, the GEL library may ask the application (several times) to supply the `TYPE` and the `SIG` belonging to a `gel_type`.

In Section B.1, we will describe the declarations pertinent to both reading and writing, in Section B.2 we will document the reading interface, and in Section B.3 we specify the writing interface.

B.1 General functionality

The file `<gel.h>` provides the following C types and macros:

```
typedef void *gel_node;
typedef long gel_type;  /* but not 0 */
#define VARYADIC -1
```

Usually, the tools will define private structures for nodes and type information, and use casts to convert to and from `gel_node` and `gel_type`. Arities are passed as integers, with `VARYADIC` denoting varyadic arities.

There is no predefined notion of characters in GEL. Because characters are used often in practice, the GEL library defines the following functions for the manipulation of 8-bit characters:

`char * char_name(int c)` Returns as a string a standard name for a character type ("'c", where c is the character). The names are statically allocated, so this function is cheaper than the obvious one-liner.

`bool is_char_name(char *s)` Checks if a string is a standard character name.

`int char_code(char *s)` Converts a character name into a character code.

Some tools might need a table to record the correspondence between gel_types and type/signature combinations. If the types are C strings (ended by a NULL character), the following types and functions can be used for this purpose:

`typedef void *gel_table` Declare a variable of this type to hold the table, with initial value NULL. Allocation will be performed by the library.

`void gel_table_insert(gel_table table, char* type, gel_type gtype)` Insert into `table` the association between the full type-name `type` and gel-type `gel_type`. Note that `gel_type` should not be 0.

`gel_type gel_table_get(gel_table table, char *type)` Retrieves the `gel_type` associated with the full type-name `type`. Returns 0 if `type` is not present in the table.

`void gel_table_cleanup(gel_table table)` Reclaim space occupied by `table`.

Both `gel_table_insert` and `gel_table_get` use time linear in the length of the full type-name.

Finally, for the implementation of a writer without using the library functions described in the next section, some basic functions for writing binary encodings of numbers and byte-sequences (see Appendix A) both to buffers and to files are supplied:

`void gel_bc_nat(long,FILE*)` writes a number to file.

`char *gel_bc_snat(long,char*)` writes a number to a buffer, and returns a pointer into the buffer, just after the last byte written.

`void gel_bc_str(int sz, char *bytes, FILE *f)` writes the length-encoded byte-sequence, length `sz` bytes, to file.

`char *gel_bc_sstr(int sz, char *bytes, char *buf)` writes the length-encoded byte-sequence, length `sz` bytes, to a buffer. Returns a pointer into the buffer, just after the last byte written.

B.2 Using the GEL reader

There are two functions for reading a GEL text, one for reading from file, the other for reading from a buffer:

```
gel_node gel_read(FILE*,gel_rcfg)
gel_node gel_sread(char*,int,gel_rcfg)
```

The first argument of `gel_sread` is the buffer, and the second argument is the number of characters in the buffer. The last argument of both functions is a pointer to a structure containing pointers to some application-specific functions (a GEL Read ConFiGuration, hence the name of the `C` type),

```
   typedef struct _gel_rcfg {
           gel_type (*define)    (int,int,char*);
           gel_node (*build)     (gel_type,int,gel_node*);
/* OPT */  void     (*set)       (gel_type,gel_node,int,gel_node);
/* OPT */  void     (*error)     (char *msg);
   } *gel_rcfg;
```

where `/* OPT */` indicates that the `set` function is optional. Undefined function pointers should be set to `NULL`.

The application-specific callback functions should do the following:

`gel_type define(int ar, int tsize, char* type);` Returns the `gel_type` with external type representation `type` of size `tsize` and arity `ar`.

`gel_node build(gel_type i, int n, gel_node child[]);` Returns a node with arity `n` and children `child[0]`...`child[n-1]`. `i` tells what type of node should be built. If some child `x` is a forward reference (eg. cycle), `child[x]` is `NULL`, and later the call `set(i,t,x,s)` will be done, where `t` is the node returned by this instance of `build` and `s` is the real son.

`void set(gel_type i, gel_node t, int n, gel_node s); /* OPTIONAL */` Set child `#n` of term `t` to the term `s`.

`void error(char *msg); /* OPTIONAL */` Print the error message, and clean up.

B.3 Using the GEL writing algorithm

There are two functions for writing a GEL text, one for writing to file, the other for writing to a buffer (which returns the number of characters written).

```
void gel_write(FILE*,gel_node,gel_wcfg);
int  gel_swrite(char*,gel_node,gel_wcfg);
```

The first argument is either the file or the buffer to write to, the second argument is the root node of the graph, and the third argument is a pointer to a structure containing pointers to application-specific functions (a GEL Write ConFiGuration, hence the name of the `C` type),

```
   typedef struct _gel_wcfg {
           gel_type (*type)      (gel_node);
           int      (*arity)     (gel_type);
           int      (*tsize)     (gel_type);
           char    *(*trepr)     (gel_type);
           int      (*size)      (gel_node);
           gel_node (*child)     (gel_node,int);
/* OPT */ void     (*error)     (char*);
/* OPT */ int      (*unshare)   (gel_node);
/* OPT */ int      (*compress)  (gel_node);
   } *gel_wcfg;
```

where `/* OPT */` indicates that a function is optional. The callback functions in this struct should be implemented as:

`gel_type type(gel_node t);` return the `gel_type` of a node.

`int arty(gel_type t);` return the arity of a `gel_type`, where the macro `VARYADIC` is defined -1 to mean varyadic arities.

`int tsize(gel_type t);` return the size of the byte representation of the type. If the representation of a type is a string (not containing `NULL` characters) this can be implemented as `strlen(trepr(t))`.

`char *trepr(gel_type t);` return the address of a buffer containing the byte representation of a type. In many cases, this will simply be an ASCII string.

`int size(gel_node t);` return the actual number of children of `t`.

`gel_node child(gel_node t,int i);` return the `i`th child of t.

`void error(char* msg); /* OPTIONAL */` Print the message `msg` and clean up.

`int unshare(gel_node n); /* OPTIONAL */` if defined, should return non-zero value if it is allowed to unshare the node `n`.

`int compress(gel_node n); /* OPTIONAL */` if defined should return non-zero value if it is allowed to compress the node `n`.

B.4 A complete example of a tool

A simple example of the use of our library is given below. This example is able to read and write binary trees with internal nodes typed `AND`, and leaves typed `TRUE` or `FALSE`.

```
#include "gel.h"

/* codes for the node types */
#define TRUE  1
#define FALSE 2
#define AND   3

/* correspondence table of types and names */
gel_table bool_types;

/* initialize table */
void init_table()
{ gel_table_insert(&bool_types,"TRUE",TRUE);
  gel_table_insert(&bool_types,"FALSE",FALSE);
  gel_table_insert(&bool_types,"AND",AND);
}

/* define a structure for nodes */
typedef struct _node {
        gel_type code;
        gel_node *children;
} *node;

/* functions for reading */
static void error(char *msg)
{ fprintf(stderr,"%s\n",msg);
  exit(1);
}
```

```
gel_type define(int ar, int tsize, char* type)
{  gel_type t;
   if (t= gel_table_get(&bool_types,type)) return t;
   else error("unknown type");
}

gel_node build(gel_type t, int ar,
               gel_node child[])
{  node n = (node)
     calloc(1,sizeof(struct _node));
   switch(t) {
     case TRUE: n->code = TRUE;
                return (gel_node)n;
     case FALSE: n->code = FALSE;
                 return (gel_node)n;
     case AND: n->code = AND;
               n->children = (gel_node *)
                  malloc(2*sizeof(gel_node));
               n->children[0] = child[0];
               n->children[1] = child[1];
               return (gel_node)n;
     default: error("unknown type");
   }
}
```

```
static struct _gel_rcfg RC = {
  define, build, NULL, error
};

/* functions for writing */
static gel_type type(gel_node n)
{return ((node)n)->code;}

static int arity(gel_type t)
{ switch(t)
  { case TRUE:  return 0;
    case FALSE: return 0;
    case AND:   return 2;
    default: error("internal error");
  }
}

static int tsize(gel_type t)
{ switch(t)
  { case TRUE:  return 4;
    case FALSE: return 5;
    case AND:   return 3;
    default: error("internal error");
  }
}

static char *trepr(gel_type t)
{ switch(t)
  { case TRUE:  return "TRUE";
    case FALSE: return "FALSE";
    case AND:   return "AND";
    default: error("internal error");
  }
}

static int size(gel_node n)
{ return arity(((node)n)->code); }

static gel_node child(gel_node n, int c) {
  switch(((node)n)->code) {
    case TRUE: error("no children");
    case FALSE: error("no children");
    case AND: if (c<2) return ((node)n)->children[c];
              else error("wrong index");
    default: error("internal error");
  }
}

static struct _gel_wcfg WC = {
  type, arity, tsize, trepr, size,
  child, error, NULL, NULL
};

main()
{ gel_node n;

  /* initialize */
  init_table();

  /* first read a graph, then write it */
  n= gel_read(stdin,&RC);
  gel_write(stdout,n,&WC);

  exit(0);
}
```

C. Full-Gel-syntax

Module Full-Gel-syntax
imports Gel$^{(7.1)}$
exports
lexical syntax

$\sim[\backslash n]+$	→ TYPE
"$\backslash\sqcup$"	→ SPACE
$['a\text{-}zA\text{-}Z0\text{-}9]$	→ SHORT-TYPE
"$\backslash n$"	→ RET

Scanners generated from SDF definitions always prefer non-Layout. Effectively, the very liberal lexical sorts defined in this module would prevent the recognition of Layout between the equations of other modules. Therefore, it is not possible to define these lexical sorts in the module Gel-types.

D. Layout

Module Layout
exports
lexical syntax

$[\sqcup\backslash t\backslash n]$	→ LAYOUT
"//"$\sim[\backslash n]*[\backslash n]$	→ LAYOUT
"%%"$\sim[\backslash n]*[\backslash n]$	→ LAYOUT