Aspects of Large Scale Symbolic Computation Management

Joachim Apel
Universität Leipzig, Institut für Informatik
Augustusplatz 10–11, D–04109 Leipzig, Germany
E-mail: apel@informatik.uni-leipzig.de

Uwe Klaus
Hochschule für Grafik und Buchkunst Leipzig
Wächterstr. 11, D–04107 Leipzig, Germany
E-mail: uklaus@hgb-leipzig.de

Abstract

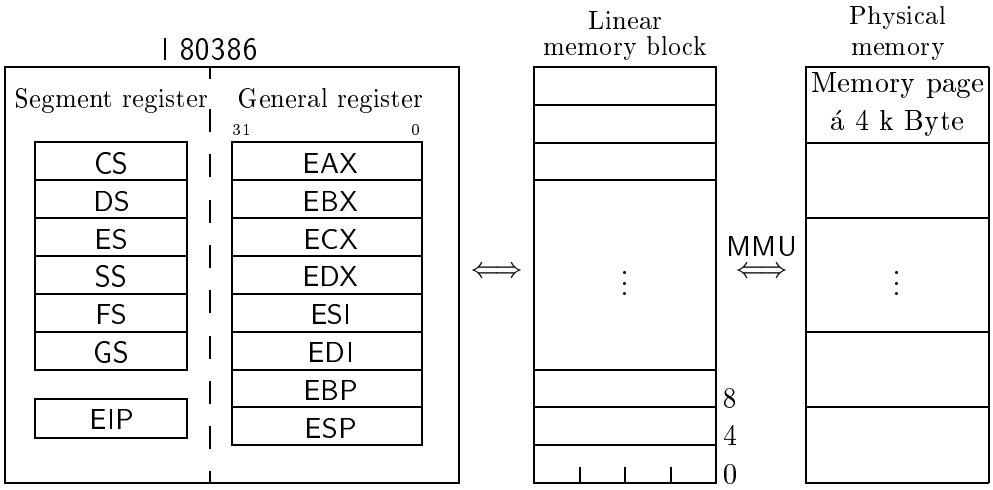
The special-purpose computer algebra system FELIX is designed for computations in constructive commutative and non-commutative algebra. In this paper we discuss some features of the system supporting the computation of rather complex problems, especially standard basis computations, using standard hardware. There is a first aspect concerning the definition and implementation of the basic data types which should be a good compromise between space and time efficient representations of the algebraic objects. Usually, rather complex computations are very time consuming (up to weeks) and often require several attempts. So, there are included special session saving methods in FELIX which allows to backup the attained intermediate results in form of memory images into special session files and to restart later on. Finally, we describe our efforts crunching complex problems by parallelization. The implemented interface is based on stream sockets and includes a special protocol for the data exchange. It supports the distributed computation on heterogeneous, loosely coupled systems.

1 Introduction

FELIX is specially designed for computations in and with algebraic structures and substructures. Buchberger's algorithm for the computation of Gröbner bases of polynomial ideals and its generalizations to non-commutative rings play a central role in the system. It is well-known that the complexity of the algorithm is horrible. In fact, it is already double exponential in the number of indeterminates in the simplest case of polynomial rings. Memory requirements of Mega bytes and time requirements of hours are not unusual in practical applications. Hence, the management of large scale symbolic computations is a crucial point in the design of a computer algebra program. In this paper we will report about some of the FELIX-facilities supporting large computations.

First of all, large scale computation requires powerful memory management tools. Creating and releasing intermediate objects often leads to a cleft memory after a short time. Moreover, the work with a multitasking operating system requires a moderate memory allocation strategy. The system should start with only a reasonable large amount of memory. But it should have also the possibility to enlarge its data segment

Figure 1: Linear memory model



later on if necessary. Furthermore, it may happen that an intermediate calculation requires much memory which will be not needed afterwards. So, the system should be fair enough to release some of its allocated memory for redistribution by the operating system. Section 2 of this paper will explain the memory management tools of FELIX.

A second feature which is very convenient for large scale computations is a debugging mode. A special signal causes the system to start a debugging shell. The user may examine the sequence of called functions including their parameters and local variables. Furthermore, he may execute an arbitrary number of statements, e.g. he can reassign variables and redefine functions. In particular, reassigning parameters or local variables makes only sense during the development of algorithms. The user is responsible himself for the consequences of his manipulations. The debugging mode will not be subject of this paper. Only its use for saving the state of a session will be sketched in Section 2.

Furthermore, the possibility to compile functions is very important to speed up computations. FELIX has got an in-build compiler which is explained in [AK93].

Finally, a promising way for attacking a large scale computation is to cut it into pieces which are solved in parallel. Section 3 informs about the parallel concepts implemented in FELIX. The concept is based on the Berkeley UNIX 4.xBSD interprocess communication facilities. More precisely, messages are exchanged according to the Internet protocol TCP/IP via stream sockets. A byproduct of the parallel concept is a fast machine independent saving mechanism for selected data.

2 Basic data types and memory management

Current multitasking operating systems usually support a linear memory model for application programs. This model is common in use and simplifies the implementation over several platforms. Figure 1 underlines the situation for systems based on Intel 32 bit processors (I 80386/486, Pentium). The segment selectors are fixed by the operating system and select a linear memory block. The processor's in-built memory managing unit (MMU) maps this block pagewise to the physical memory or

additional swap space. The application program can only modify the general purpose register without restrictions. These registers are of machine word size and contain either signed/unsigned integers or pointers/offsets relative to the selected linear memory block. So, an important designing principle of FELIX was that every algebraic object independent from its size and complexity can be identified by a single machine word.

The basic data types inside FELIX strongly reflect properties of the represented algebraic objects. Any data is either an atom or a sequence of other data (lists). So, the general structure is LISP–like, but there are additional atomic data types which are particularly adapted to our problem classes:

- *names*,
- *integers*,
- *rational numbers*,
- *character strings*,
- *exponent vectors*,
- *bitfields*, and
- *packed lists/arrays*.

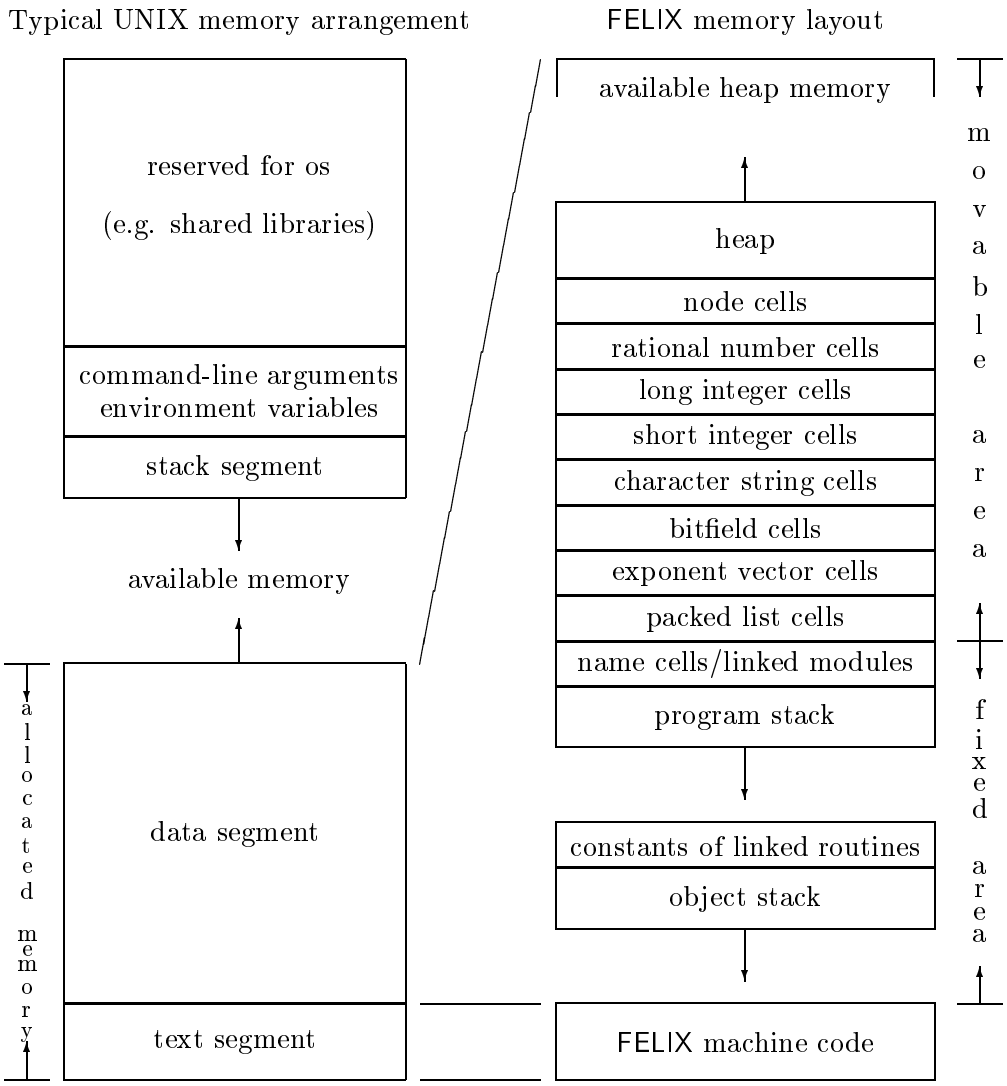
A consequence is that in contrast to many LISP–implementations the storage model of FELIX is inhomogeneous, i.e. different data types are stored in different storage areas (see figure 2). For the typical layout of UNIX programs we refer to [St92].

A name cell contains three different values simultaneously. Depending on the syntactical context of a name we may refer to:

- its value as a variable with dynamic binding according to the sequence of function calls,
- its global property (similar as in LISP), or
- its operator definition.

Note, that there are also local variables with static binding according to the sequence of function calls. But after parsing an operator definition the references to the concrete names are replaced by such to special virtual (non-name) objects maintained only on the runtime object stack. The implementation of *integers* distinguishes between short and long numbers. Shorts are within the range of a machine word and stored directly in their cells. The data type of *exponent vectors* was created to support a commutative polynomial arithmetic. It is based on a sparse representation of the exponent vector of a monomial. Within FELIX there are included sixteen machine routines which are suitable for an efficient implementation of a polynomial arithmetic. The *bitfields* correspond to the non-commutative case. A non-commutative monomial is stored as a sequence of integers representing the ring indeterminates. These integers are coded with some bits only since the number of ring indeterminates is usually small. *Long integers*, *character strings*, *exponent vectors*, *bitfields*, and *packed lists*, which correspond to data of variable size, are represented by two parts: a cell where they are registered, and a heap entry where their elements are stored (see figures 3, 4). A more detailed description of the internal representations can be found in [AK94].

Figure 2: FELIX memory map



Complex objects are constructed either as binary trees by node cells in the usual LISP–like way or by arrays, the so called packed lists.

There are two different kinds of garbage collections. The first one is activated if no cells are available. It is performed in a usual way. First, there are marked all the current occupied data beginning with the initial data, the name cells, the temporary computed elements on the runtime object stack, the constants of the linked modules, etc. and then recycled all the unused cells.

The whole available memory (see figure 2) may be used to extend the heap. If a new heap entry is requested it will be stored immediately behind the heap and the pointer FREEHEAP at the begin of the available memory will be updated. It may happen that the remaining available memory is too small to create a new entry, although, lots of memory are unused. For this reason, there is a second kind of garbage collection to

Figure 3: Cell structure

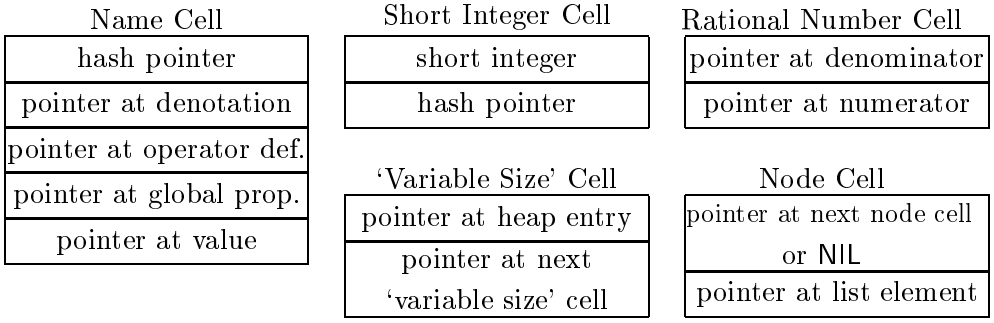
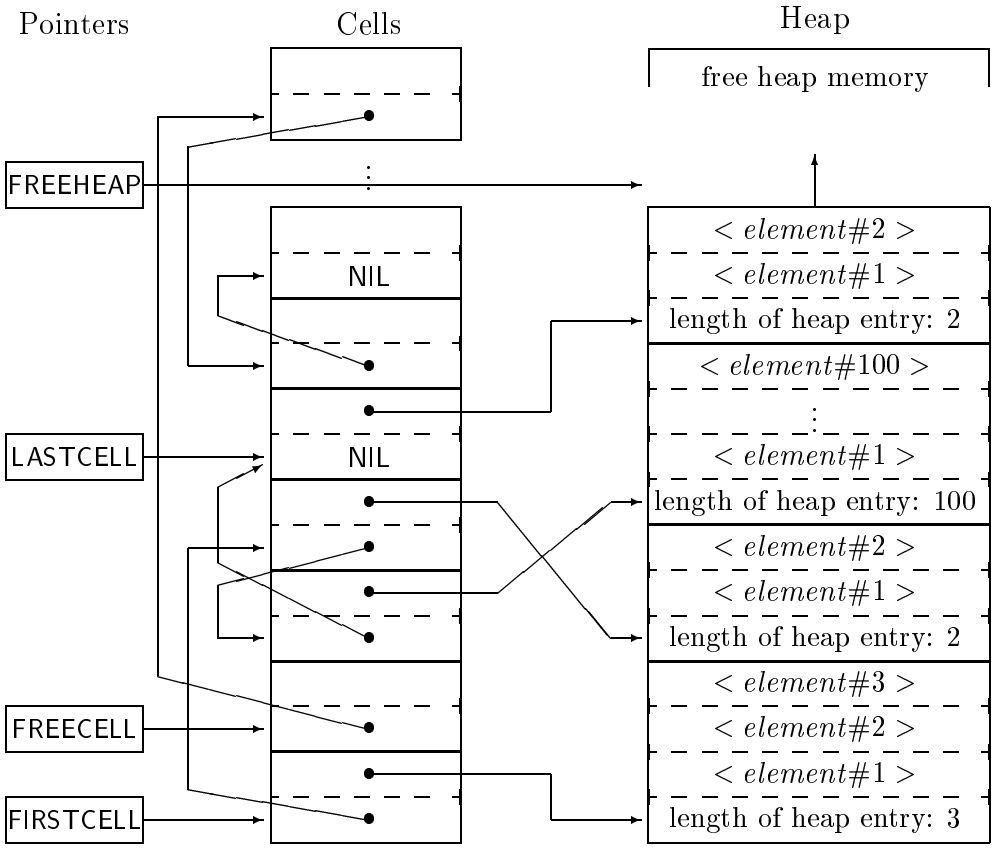


Figure 4: Heap management



contract the heap. Since the chain formed by the first pointers of all 'variable size' cells reflects the time of creation of the heap entries, passing through this chain causes working off the heap linearly from the bottom to the top. This enables contraction without managing gaps (see figure 4).

A combined contraction and movement of the heap towards the memory end pro-

Table 1: Conditions for allocating new cells

condition	zone 1	zone 2	zone 3	zone 4
number of free cells is less than a predefined minimum	$\surd$	$\surd$	$\surd$	$\surd$
the shortage of the same kind of cells was the cause for this and the last garbage collection	$\surd$	$\surd$	—	—
ratio $\frac{\text{number of free cells}}{\text{number of allocated cells}}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	—

Table 2: Conditions for allocating additional heap memory

condition	zone 1	zone 2	zone 3	zone 4
free heap size is less than a predefined minimum	$\surd$	$\surd$	$\surd$	$\surd$
ratio $\frac{\text{size of free heap memory}}{\text{size of used heap memory}}$	$\frac{1}{1}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$

vides the necessary space to extent the number of cells of any sort. So, cells can be created at that time when they are requested. If the free heap memory is insufficient to satisfy some request after a heap contraction the data segment size is enlarged by the UNIX *sbrk(int incr)* system call.

Since FELIX is designated to run in a multitasking environment we worked out some heuristics to ensure that the system is allocating only as much memory as necessary, and not as possible. Depending on the amount of the physical memory we introduced four memory boundaries. Exceeding such a boundary will increase the conditions for cell or heap memory allocation.

- Boundary 1: initial and minimal allocated memory (5–20 % of physical memory)

  The FELIX session can coexists with other memory intensive programs well.

- Boundary 2: about half of physical memory

  The FELIX session dominates the system and should be combined only with I/O–intensive programs.

- Boundary 3: 15 % below physical memory

  No other applications except necessary operating system stuff is possible.

- Boundary 4: all the physical memory

  Further success can not be expected. The user should save the state of the computation for further attempts on stronger hardware and give up.

In dependence of the size of the data segment (the zone between two of the boundaries) there will be performed separate checks for each cell type after each (cell) garbage collection. Table 1 shows the kind of checks taking place within the four zones. The sign — means that a check is skipped since allocation of new cells is disabled in order to

Table 3: Condition for performing full cell contractions

condition	zone 1	zone 2	zone 3	zone 4
number of garb. collect. forcing a contraction	2048	1024	512	256

avoid enlarging the data segment too much. The sign $\surd$ stands for enlarging the corresponding cell area in case of satisfied checks. If the ratios fall below the fractions given in the table then new cells will be allocated. Similar remarks are valid for enlarging the heap memory after (heap) contractions (see table 2).

Nevertheless, the situation may happen that first a certain kind of cells is extensively allocated and later on lots of unused cells occupy the data segment while the demand for other cells is growing. Therefore we introduced a special type of garbage collection capable of contracting the cells such that all currently bounded cells are moved to a continuous memory block. These (cell) contractions are regularly performed after every n-th garbage collection (see table 3). At the same time a certain upper part of the free heap memory is released.

The FELIX session saving concept also takes advantage of this technique. The inbuilt function *save* called by

$$save(file\_name)\$$$$

performs such a contraction and writes the data segment without the free heap memory (compare figure 2) into the session save file *file_name*. Obviously, the contraction saves lots of disk space. Furthermore, information about open input and output files is kept in the session save file. The computation can be restarted later on by calling

$$load(file\_name)\$.$$

Output files are opened in append mode and input files are opened for reading and repositioned to the offsets at saving time according to the special information about open files.

There is also an interface to the UNIX signal facilities (see [L90]). Receiving the signal SIGQUIT interrupts the computation and gives the user the opportunity to interact with the system (e.g. to debug or to save the session). Also, the occurrence of SIGUSR1 and SIGUSR2 can be handled by user defined functions. These signals can be delivered user independently using e.g. the `crontab` service and session saving may be performed automatically.

3 FELIX–interprocess communication

We start with a rough description of the parallel concept implemented in FELIX based on the Berkeley UNIX 4.xBSD interprocess communication facilities. Among the various possibilities provided by Berkeley UNIX 4.xBSD our choice was that each process separately creates its socket and that the communication is performed via the Internet protocol TCP/IP (see [St94]) since this is a common method allowing the communication between processes running at different machines. Furthermore, we preferred the

use of streams rather than datagrams, i.e. we establish a permanent connection between both sockets which is used for the data transfer. The main argument for the stream philosophy is the fact that a stream works according to a reliable first-in/first-out strategy. So, the master process may feed the slaves without waiting for a confirmation of the reception. Hence, the overhead for stream controlling is less than for datagrams from the view point of the programmer. The more frequent the transfer rate the more preferable is our decision. A further important advantage is that a stream appears to the programmer as a sequential file which enabled us to use the parallel mechanisms also for storing intermediate results of large scale computations. A disadvantage of the stream concept is that it is more susceptible to network errors since broken pipes cannot be repaired but has to be reinstalled. For an overview about the interprocess communication types under Berkeley UNIX 4.4BSD we refer to [S94].

We summarize, two FELIX-processes A and B may communicate via stream sockets using the Internet protocol TCP/IP. Establishing a connection is asymmetrical. Each of the processes A and B creates a socket but then one of the processes, let us say A, will be the server and the other, i.e. B, the client of the connection. The server will provide a port for the connection and then wait for the clients reply. The client has to know the port address in order to build up the connection and to inform the server about the successful installation. The connection is a (duplexed) channel between A and B which consists of two one-way communication mechanisms called pipes. There is a pipe (A, B) which serves A for writing and B for reading and a second pipe (B, A) with the opposite meaning for A and B. The processes A and B can access the channel via handle numbers a and b, respectively. Note, that both pipes (A, B) and (B, A) are referred by the same handle number within the same process. Which pipe has to be used gets clear from the context. After establishing a channel the meaning of A and B is not longer asymmetrical. Both processes can read from and write to the channel, and watch the channel for events. At the UNIX level the data transfer is byte oriented. In order to simplify the programmers work with FELIX the systems provides read and write operations which are FELIX-object oriented. Later we will describe the details of the underlying protocol for the data exchange. Finally, each process can close its own channel end. This will not affect the socket of the other process but, of course, the channel will be destroyed and one has to ensure an appropriate error handling of the second process after the next (unsuccessful) access attempt. The same error handling will be invoked by both processes if for some other reason the channel was broken.

3.1 Setting up a stream socket connection

Let A and B denote two FELIX-processes. The creation of a channel connecting A and B requires the distinction between *server* and *client*. Let us say: we install A as server and B as client of the connection.

Figure 5 shows the instructions necessary for opening the channel at the server side. The function call

$$socket()\$$$

creates a new socket and returns a positive integer *soc* which is the handle of the socket. Note, however, that this handle number is only of intermediate importance. As soon as the connection is established another handle will be used for the communication. The second statement effects that the process will be the server of the channel ending

Figure 5: Installing a server A

$$soc := socket()\$$$
$$por := server(soc)\$$$
$$a := accept(por)\$$$

Figure 6: Installing a client B

$$b := socket()\$$$
$$client(b, inet, por)\$$$

at socket *soc* and returns the port number *por* used for the connection. Finally, the function call

$$accept(por)\$$$

causes the process to wait for a confirmation by the client that the connection is established. The resulting positive integer a is the handle number for the access to the input and output streams belonging to the channel.

Let us consider now figure 6 describing the connection establishing process from the client side. Here the first statement will again create a new socket and return its handle number b. But in contrary to the server this handle number will be used later for the communication. The second statement effects to connect the handle b to port *por* at the machine with the Internet IP number *inet*. If the connection could be established the result of the function call will be TRUE and FALSE, otherwise. Only, afterwards the *accept*-function call in figure 5 can be finished.

We see that the client has to get informed about the Internet number of the partner process and the associated port, i.e. the name bound to the server socket (see [S94]). Establishing channels for the communication between FELIX-processes working in parallel by applying only these tools would be rather tedious. One would have to start the FELIX-sessions on both machines[1] and then to perform step by step the above instructions for installing server and client. In order to make the work more comfortable we implemented a function *connect* which creates a slave process[2] which can be fed with tasks to be solved in parallel to the master. The *connect*-function is called by the server process with exactly one parameter, namely, a string of the form *host_os*. The first part *host* of the string is the name of the host machine where the client is to be started. The second part *os*, separated by the character _, specifies the operating system running on the host machine. Up to now the following operating system tags are allowed: *SPC* for the Sparc system on Sun workstations, *BSD* for NetBSD, and *LNX* for Linux both running on IBM-compatible personal computers based on a 32 bit processor. The user needs to have an account on the host machine and he has to

[1] Of course it is also allowed to start both sessions at the same machine.

[2] Note, that this is not a child process in the sense of UNIX, since it is not a copy of the original process but a new started FELIX-session with a particular initialization.

ensure by editing the file `.rhosts` that he is allowed to start a remote shell on this machine without being asked for a password. The action of the function call

$$connect(host\_os)\$$

can be roughly, i.e. ignoring any details concerning safety aspects such as avoiding dead locks in case of failing connections, described as follows. There are performed the first two statements of figure 5. Then it is called a operating system remote shell which executes the UNIX-command:

```
rsh <host> "(FXMODULE=<fxmod>;export FXMODULE;
             FELIX=<fxclient>;export FELIX;
             SERVER=<server>;export SERVER;
             PORT=<por>; export PORT;
             cd <fxclient>;
             fxexe)" &
```

The names enclosed in angle brackets are place holders for the following parameters:

`<host>`	stands for the name of the host machine in a form resolvable by the name server of the server machine.
`<fxmod>`	denotes the directory where the FELIX module files are located at the host machine.
`<fxclient>`	denotes the work directory where FELIX has to be started in the client mode at the host machine.
`<server>`	denotes the name of the server machine.
`<por>`	is the number of the port which will be used for the connection to be established.

The parameter `<host>` is taken from the actual parameter of *connect*. `<por>` is the port number resulting from statement two of figure 5. The parameter `server` is computed using the UNIX-command `hostname` during each session start and remains global knowledge during the whole session. The value of the parameters `fxmod` and `fxclient` are computed depending on the operating system tag *os* contained in the actual parameter of the *connect*-call. The execution of the above UNIX-shell-command will start a new FELIX-session on the host machine. Because of the parameter `fxclient` the session start will include the statements of figure 6. FELIX has got a function *environment* which allows to access the UNIX environment. By that means it has access to the name of the server machine and the port number provided for the connection. The initialization of a client session includes the construction of a list of allowed servers. Using this list the Internet number belonging to the server machine can be resolved. Note, that the above shell command is running in parallel to the FELIX-server session. If the start of the command was successful the function *connect* will continue its work with performing statement three of figure 5. The result of the function will be the handle number a serving for the communication with the new created client.

Note, that compared with the step-by-step method a big advantage of the function *connect* is the much easier handling and that the client session can run without having an output window. As a disadvantage one could consider the fact that it is impossible to work directly with the client. Actually, the behaviour of server and client will remain asymmetrical also after establishing the channel.

Our performance strategy of a client does not bind its life time to the solution of one particular task. In order to fulfil the requirements on a slave a client-FELIX-session has another interpreter loop as a usual FELIX-session. We have a cyclic performance of

Watch_channel – Read_channel – Eval – Write_channel

in contrary to the usual cycle

Input – Eval – Output .

Sending the object ($'bye$) to the client will cause to finish the client session[3]. Afterwards, the server should release also its own handles.

3.2 Data transfer across a channel

Let *hdl* be the handle number of a client or server process which addresses a channel for the communication with the partner.

There is a function *rawread* called by

$$rawread(hdl)\$$$

which takes *hdl* as its only argument. The semantics of this function call is to read one FELIX-object from the input stream belonging to the channel addressed by *hdl* and to return it as result. But, note, if the input stream is currently empty then the session will be blocked and wait until an object arrives.

A second function *rawwrite* allows to send a FELIX-object to partner processes. Sometimes information has to be distributed to more than one partner. The transfer of an object requires to convert it according to the communication protocol. In order to avoid multiple conversions the function *rawwrite* may feed more than one client simultaneously. Let n be a positive integer which is at least as large as the largest handle number to be served. It is created a packed field (array) *ports* of length n and assigned the value TRUE to the $(i+1)$-st entry of *ports* if the channel with handle i[4] should receive the object and to FALSE, otherwise. The call

$$rawwrite('ports, obj)\$$$

will transmit the object *obj* to all handles addressed in the above way by setting the corresponding entry of *ports* to TRUE. The result of the function call is the number of processes for which the transfer was successful, i.e. where it was possible to write to the corresponding stream but it is not tested whether the stream can be read successfully. As a side effect the packed list *ports* is manipulated in such a way that only the entries corresponding to channels where the write operation was successful have the value TRUE and all other the value FALSE. So, the programmer can control the state of the clients and perform an error handling if necessary. Note, that *rawwrite* will block the process if a stream buffer is full and will wait until it can finish the write operation. Also any receiving process will be blocked as soon as it tries to read the object from its channel.

Finally, the function *multiplex* allows to watch channels. The function call has the syntax

$$multiplex('ports1,'ports2,'ports3, t)\$$$

[3]This includes to close all open handles, hence, also the socket is closed automatically.

[4]The range of handles starts with 0.

The meaning of *ports*1, *ports*2, *ports*3 is the same as for *rawwrite*. The first array declares the channels to be watched for incoming messages, the second array tests whether the output pipes are ready for writing and the third array corresponds to exception events. One or two of the arrays may be replaced by the name NIL which abbreviates that no channels should be watched for events of the corresponding type. So the most important application is to replace *'ports*2 and *'ports*3 by NIL but we decided to pass all possibilities provided by the Berkeley UNIX system call *select* to the programmer. The last argument t defines a watching duration. If t takes the value NIL then the process is blocked until at least one event occurs. But t can also be a natural number which gives the maximal duration in milliseconds. In particular, if we watch for incoming messages and use $t = 0$ then we can check whether there is some non-empty input stream (among the selected channels), i.e. some unread object, or not. At latest, if there occurs a controlled event the function will finish its work and return the number of registered events. Note, that e.g. an incoming message will be registered by any *multiplex*-call controlling the corresponding input channel until it had been read using *rawread*. Again we have the side effect that the entries of *ports*1, *ports*2, *ports*3 corresponding to a registered event are updated. A desirable feature of *multiplex* would be to test whether an output stream buffer is empty enough for keeping the next object to be transferred. However, this is impossible. So it is not always possible to avoid a jam in some stream which will block the writing process until the partner process will remove objects from the stream.

3.3 Transmission protocol

Recall the fact that the functions *rawread* and *rawwrite* work FELIX-object oriented. In Section 2 we mentioned that a FELIX-object can be an atom or a list. Furthermore, in that section we described some details concerning the internal representation of atoms. In addition, we have the data type of *node cells* which allows to build up lists. A node cell consists of two pointers to other FELIX-objects. The FELIX-object starting at a node cell is a binary tree. Its left (first) branch is the object addressed by the first address of the cell and its right (rest) branch is the object where the second address refers to. This is the usual LISP–philosophy for building up complex objects[5]. In contrary to LISP we restricted the class of constructable binary trees by considering only genuine lists (i.e. the right most branch is NIL). Note, however, that using destructive functions which can replace the first respectively the second branch by any other object, in particular the object starting at the node itself, the user might construct objects which contain cycles.

So we summarize, a FELIX-object may be an atom or a list, i.e. a sequence of "smaller" objects. There are functions *ltop* and *ptol* which convert a list to a packed list (array) and vice versa in the usual way. Note, that FELIX allows also arrays of length zero, i.e. with no entries.

Atoms of the types string, vector, and bitfields are transmitted by sending first one byte which is a tag characterizing the atom type and afterwards word by word the heap entry belonging to the atomic object. Note, that the first word always contains the information about the number of remaining words of the heap entry according to the internal representation of a variable size object on the heap shown in figure 3.

[5]The first branch is the CAR and the rest branch the CDR of the object.

Table 4: Type tags used for the data exchange protocol

Object type	Tag character
String	"
Vector	[
Bitfield	\
Short integer	0
Long integer	1
Rational	/
Name	@
Packed list	{
(Node) list	(

Since the internal representation of integers (byte order) depends on the underlying processor the data exchange between different platforms requires the conversion to a unique code. Our choice was the *big endian*[6] code. On Sun workstations no conversion is necessary but on Intel-based architectures (little endian) we have to convert integers before sending and after receiving them. So, a short integer will be transmitted by sending the tag character 0 followed by the big endian code of the integer. A long integer will be transmitted by sending the 1-tag, the number of digits (of word size) and finally digit by digit converted to the big endian code. A rational number will be send as the tag / followed by first numerator and second denominator, where both of them are either short or long integers encoded as described above. A name object is transmitted as the tag @ followed by a byte giving the number of characters belonging to the denotation of the name and then character by character the denotation. Note, that only the denotation and not the entire object is subject of transmission.

The transmission of packed lists and lists formed by nodes is performed recursively. Before the transmission a list is converted to a packed list. After the tag characterizing the type it follows the number of entries and then entry by entry the objects of the sequence. Each object is decoded itself according to the protocol.

Table 4 shows all data type tags used for the protocol. According to the protocol a receiving process is able to reconstruct the arriving object. In particular, the number objects and the lists formed by nodes are converted back to the appropriate internal representation.

The protocol is based only on FELIX-object conventions. Implementation depending facts as integer representations or even addresses have no influence on the transmitted data. Clearly, the transfer of absolute addresses makes no sense even for processes running on the same platform since we do not support shared memory access. Our philosophy has the following inconvenient side effect. Complex objects (lists and packed lists) can share subobjects, i.e. they can refer directly or indirectly to one and the same smaller object. After transferring both objects to another process via a channel this property will be lost, in general. If the common subobject is an atom different from a packed list then the unique data representation will help (see [AK94]) but if the subobject is a packed list or a node list then only structural equality is preserved.

[6]Most significant byte is stored first.

Figure 7: Storing selected data

$$
\begin{aligned}
&create(\texttt{"x.sv"})\$\\
&hdl := open(\texttt{"x.sv"}, 1)\$\\
&por := consplist(hdl + 1)\$\\
&setplist(por, hdl + 1, true)\$\\
&rawwrite('por, x)\$\\
&rawwrite('por, list('assign, list('', 'x), x))\$\\
&close(hdl)\$
\end{aligned}
$$

Hence, tasks which make essential use of the physical equality, i.e. the equality of addresses, of objects cannot be passed to other processes. The user should be also aware the fact that for the same reason data sometimes will occupy more memory in the receiving process than it did in the producing process.

3.4 Saving data via the FELIX-interprocess communication protocol

In Section 2 we mentioned the FELIX-facility to save the whole state of a session into a file. However, note that the session must not have open sockets for interprocess communication. Later such a file can be loaded very fast in another session and the execution can be continued under the same conditions as before. Since all offsets are relative it is possible to run the save-file on any platform which has the same processor and the same version of the operating system as at saving time. Nevertheless, save-files are machine dependent, e.g. it is impossible to load a file under Linux which was saved under Solaris.

Therefore, FELIX supports a second saving mechanism. It allows to write selected objects into a file. In contrary to the session save mechanism the input of this file will not restore the whole state of the old session. But, it will also not destroy the settings of the new session which are independent of the saved data. Recall that one argument for our decision for stream sockets was that the same concept can be applied also to writing data to and reading data from ordinary files in a machine independent representation which is very closed to the internal representation.

Let us consider the following example. We want to store the value of the variable x to a not yet existing file with name `x.sv`. The solution is shown in figure 7. After creating the file `x.sv` it is opened for writing. It is constructed a packed list with all entries FALSE at initial time. Then the entry corresponding to the handle *hdl* is set to TRUE. It may follow an arbitrary number of *rawwrite* operations which will write the second argument to the file according to the conventions of the protocol described in the preceding subsection.

Figure 8 shows how the stored data can be restored later on. The file `x.sv` is opened for read access which will place the file pointer automatically at the begin of the file. Then the objects are read in from the file. Note, that only the value of x without information about the name x itself is stored during the first output operation. So the variable x will not be influenced by the first *rawread*-call presented in figure 8. But

Figure 8: Loading selected data

$$hdl := open("x.sv", 0)\$$$
$$y := rawread(hdl)\$$$
$$eval(rawread(hdl))\$$$
$$close(hdl)\$$$

the input of the second object will yield the appropriate assignment instruction. Its evaluation will cause the assignment of the old value of x to the variable x.

Note, that such a way of storing huge expressions is much more efficient than to store it in the natural FELIX-input format since the necessary conversions before the output and after the input are much less time consuming. However, loading a session save file is still much faster then reading stored data using the *rawread*-function. The main reason for this difference is the significant larger number of calls of the operating system routine *read* in the latter case.

Summarizing we can say that storing intermediate results according to the conventions of our interprocess communication protocol is machine and operating system independent but still very closed to the internal representation at an arbitrary platform.

References

[AK93] J. Apel, U. Klaus, Data Representation and In-built Compilation in the Computer Algebra Program FELIX. LNCS **721**, pp 173-192, 1993.

[AK94] J. Apel, U. Klaus, Representing Polynomials in Computer Algebra Systems. Proc. New Computer Technologies in Control Systems, Pereslavl 1994.

[L90] S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman, The Design and Implementation of the 4.3BSD UNIX Operating System. Addison Wesley, Reading, Mass., 1990.

[S94] S. Sechrest, An Introductory 4.4BSD Interprocess Communication Tutorial. In: 4.4BSD Programmer's Supplementary Documents, O'Reilly & Associates, Inc. 1994.

[St92] W.R. Stevens, Advanced Programming in the UNIX Environment. Addison Wesley, Reading, Mass., 1992.

[St94] W.R. Stevens, TCP/IP Illustrated, Vol. 1: The Protocols. Addison Wesley, Reading, Mass., 1994.