Amos II programmer's conventions

Jonas S Karlsson, Tore Risch, Magnus Werner, etc.

May 17, 2004

Abstract

Amos II (Active Mediators Object System) is a Main-Memory Object Functional Database System.

This document describes how to extend the Amos II system kernel with more primitives and functionality on the C/Lisp level along with coding conventions for this. For high level APIs to C, Lisp see documents *external.pdf* and *javaamos.pdf*. The built-in ALisp system and the internal storage manager is documented in *alisp.pdf*.

Contents

1 Introduction

2 Getting started

First section describes how to install the system with source code after it has been installed into the repository.

2.1 Installation

Your PC need a number of software installations in order for you to be able to install the development version of the system and be productive. See *http://user.it.uu.se/~udbl/setupinstructions.txt*.

The system resides on the UDBL account on IT:s file system. You need to be member of the UDBL group on IT's Unix system in order to check out the sources. Furthermore, for convenience you should have the same login names both under Unix and on the PC you are using.

The system sources are stored in a CVS-REPOSITORY and in order to do software development for the Amos II system one has to check out an own version from the REPOSITORY. See *http://user.it.uu.se/~udbl/software/cvs-instructions.html*.

Decide where your amos root directory should reside, go to this directory and checkout AMOS with the Unix/Windows commands:

```
cvs co AmosNT
```

After checking out, directory `AmosNT` will contain all source code necessary to build your own system.

2.2 Compilation

To compile and test the system. Execute in folder `AmosNT/bin` the command procedure:

```
install
```

which will compile the system. If not, some environment variable is wrong or you have not installed all necessary software on your PC. See *http://user.it.uu.se/~udbl/software/cvs-instructions.html*.

A successful full installation also includes a full regression test. Run it by typing in folder `AmosNT/regress`:

```
testmaster
```

If the regression fails you probably have not installed all software required. See *http://user.it.uu.se/~udbl/software/cvs-instructions.html*.

2.3 Checking In

For detailed informations about how to check in code changes to CVS see *http://user.it.uu.se/~udbl/software/cvs-instructions.html*.

3 Running Amos II from XEmacs

The most convenient way to develop Alisp and AmosQL code is to run Amos II from within XEmacs. Certain features of Amos II and extensions of Emacs facilitate this.

To get the Emacs extensions you should copy the file `lsp/init.el` to the .xemacs directory where XEmacs keeps its init files. This is usualy in `C:/.xemacs` or `C:/Documents and Settings/<username>/.xemacs`.

When XEmacs is started give the command

```
M-x-shell
```

This will start a new Windows (or Unix) shell inside XEmacs. You can there give the usual Windows (Unix) commands. Run there Amos II by typing:

```
amos2
```

If you are developing ALisp code, enter ALisp with the command

```
lisp;
```

Lisp function APROPOS

```
(apropos symbol)
```

`apropos` searches through the entire system to find out what ALisp functions, STRUCTs, etc. contain the string SYMBOL and prints the location of their definition along with the documentation string.

If you, e.g., are changing the system for overloading AmosQL functions you might be interested in what other functions there are to deal with *resolvents* by calling:

```
lisp 1> (apropos 'resolvent)
GET-RESOLVENTS C:/AmosNT/lsp/typecheck.lsp 148
     ""
SET-RESOLVENTS C:/AmosNT/lsp/typecheck.lsp 174
     ""
GET-MOST-SPECIFIC-RESOLVENT C:/AmosNT/lsp/bindpatex.lsp 221
     ""
RESOLVENTS-WITH-ARITY C:/AmosNT/lsp/typecheck.lsp 520
     ""
GET-RESOLVENT-ARGTYPES C:/AmosNT/lsp/typecheck.lsp 635
     ""
COMPARERESOLVENTS C:/AmosNT/lsp/typecheck.lsp 267
     ""
GET-RESOLVENT-RESTYPES C:/AmosNT/lsp/typecheck.lsp 638
     ""
GEN-DIR-LIST-OF-RESOLVENTS C:/AmosNT/lsp/bindpatex.lsp 35
     ""
MATCH-ARGS-TO-RESOLVENTS C:/AmosNT/lsp/bindpatex.lsp 114
     ""
REMOVE-GENERAL-RESOLVENTS C:/AmosNT/lsp/typecheck.lsp 430
     "Remove the resolvents in LOR that are more general than other resolvents
n LOR"
MAKE-RESOLVENTNAME-DCL C:/AmosNT/lsp/typecheck.lsp 206
     "Generate resolvent name, given generic name NAME,
argument declarations ARGDCL, and result declarations RESDCL"
MORE-SPECIFIC-RESOLVENT? C:/AmosNT/lsp/typecheck.lsp 440
     "Is A a more specific resolvent than B?"
```

```
AMBIGOUS-RESOLVENTS? C:/AmosNT/lsp/typecheck.lsp 150
     ""
ADDRESOLVENT C:/AmosNT/lsp/typecheck.lsp 153
     ""
GETRESOLVENT C:/AmosNT/lsp/typecheck.lsp 177
     "Get the resolvent of the generic function FNO, given the list of
exact matching argument types ATL.
ERRORFLG != nil => generate error if resolvent not found"
RESOLVENTS C:/AmosNT/lsp/typecheck.lsp 146
     ""
SORT-RESOLVENT-LIST C:/AmosNT/lsp/bindpatex.lsp 381
     ""
GETUNIQUERESOLVENT C:/AmosNT/lsp/fncall.lsp 515
     ""
SET-RESOLVENT-TYPES C:/AmosNT/lsp/typecheck.lsp 167
     ""
FUNCTION.RESOLVENTS->FUNCTION-+ C:/AmosNT/lsp/amosfns.lsp 558
     ""
GENERATE-RESOLVENT-LIST C:/AmosNT/lsp/bindpatex.lsp 341
     "Generates a list of all resolvents of genfn that is in the dynamic
type set of the type fargt"
MAKE-RESOLVENTNAME C:/AmosNT/lsp/typecheck.lsp 189
     "Generate name of resolvent, given generic function NAME,
argument types ARGTYPES, and result types RESTYPES"
BAGCOERCERESOLVENT C:/AmosNT/lsp/collections.lsp 469
     ""
```

If you want to look at some of the definitions of these function, place the XEmacs cursor over the file name (e.g. over `C:/AmosNT/lsp/collections.lsp`) and press `F1`. This will open the definition of ALisp function `BAGCOERCERESOLVENT` in another XEmacs buffer.

Lisp function FP

```
(fp function)
```

`fp` prints the file position where ALisp `function` is defined. For example:

```
lisp 1> (fp 'bagcoerceresolvent)
BAGCOERCERESOLVENT C:/AmosNT/lsp/collections.lsp 469
```

Place the cursor over the file name and press `F1` if you want to see its definition.

Lisp function CALLING

```
(calling function)
```

`calling` prints the file positions of the Lisp functions calling FUNCTION. For example:

```
lisp 1> (calling 'optimize-pred)
BPAT-OPTIMIZE-FUNCTION C:/AmosNT/lsp/bindpatex.lsp 166
COMPILE_PHASE2 C:/AmosNT/lsp/comppred.lsp 17
REOPTIMIZE C:/AmosNT/lsp/amosfns.lsp 661
CREATEFOREIGNFUNCTION C:/AmosNT/lsp/function.lsp 455
```

Lisp function GREP

```
(grep string)
```

`grep` searches the sources of all files loaded into the current Amos II image looking for the `string` and displays the corresponding file positions. For example:

```
lisp 1> (grep "grep")
C:/AmosNT/lsp/ref_lisp.lsp:370     ;;; What lines in what files have given string? Borland grep.
C:/AmosNT/lsp/ref_lisp.lsp  372      grep ((charstring pat)) ((charstring))
C:/AmosNT/lsp/ref_lisp.lsp  374        (if (eq (system (concat "grep -r -n -o -e \"" pat "\" " f))
C:/AmosNT/lsp/ref_lisp.lsp  378     ;;; What files have given string? Save lines in INTOFILE. Borland
C:/AmosNT/lsp/ref_lisp.lsp  380      grep ((charstring pat)(charstring intofile)) ((charstring))
C:/AmosNT/lsp/ref_lisp.lsp  383          (if (eq (system (concat "grep -r -n -o -e \"" pat "\" " f
C:/AmosNT/lsp/ref_lisp.lsp  401     (defun grep (string)
C:/AmosNT/lsp/ref_lisp.lsp  403       (mapfunctionres 'grep (list (mkstring string)) nil 'null)
```

Some useful XEmacs commands for dealing with Lisp:

- `CTRL-META-q` Place the cursor over the first '`(`' of a Lisp form and press `CTRL-META-q` to pretty print it. It is strongly encouraged to let Emacs prettyprint Lisp functions. You will discover parathesis errors.
- `META-p` and `META-n` in the shell window picks upp the previously or next ALisp form or other command typed into the shell.
- `META-->` and `META-<-` in a Lisp source buffer (arrows) moves the cursor one form forward or backwards.

4 Coding convections

In ALisp there is no real packages, but grouping of files has been done in most cases and should be done by new developers.

First when adding new functions the already existing files should be studied. If there seems to be a package for that kind of operations/data then the function should be put there. The Alisp function

```
(apropos SYMBOL)
```

Finds all Alisp functions similar to SYMBOL along with their documentation, as documented above.

4.1 Files

Naming convention for lisp module files is to suffix them with `.lsp`.

4.2 Headers

Each file has a header. In *http://user.it.uu.se/∼udbl/amos/cvs-heads.html* there are headers for most kinds of files.

This is the header for Lisp files:

```
;;; ============================================================================
;;; AMOS2
;;;
;;; Author: (c) <year> <name>, UDBL
;;; $RCSfile: coding.tex,v $
;;; $Revision: 1.8 $ $Date: 2004/04/30 10:06:57 $
;;; $State: Exp $ $Locker:  $
;;;
;;; Description: <description>
;;;
;;; ============================================================================
```

This is the C header:

```
/*****************************************************************************
 * AMOS2
 *
 * Author: (c) <year> <name>, UDBL
 * $RCSfile: coding.tex,v $
 * $Revision: 1.8 $ $Date: 2004/04/30 10:06:57 $
 * $State: Exp $ $Locker:  $
 *
 * Description: <description>
 *
 *****************************************************************************/
```

Change the `<name>` to yourself and fill in the `Description:` field to reflect the modules usage.

4.3 Naming functions

When defining ALisp callable functions in C the following naming conventions are used:

- The C function name is always lower-case and suffixed with 'fn'. For example, the Lisp function `LIST` is implemented by the C function `listfn`.
- Hyphens in Lisp function names become underscores in the corresponding C function names. For example, the Lisp function `ARG-TYPES` is implemented by the C function `arg_typesfn`.

4.4 Init- and Registering function

In order to simplify the init and register process for foreign C functions in Amos II, one is encouraged to write a `void register_X_functions()` function for each C package. This function should be responsible registering external functions using `extfunctionX` (For examples, look in: `system/C/text_fns.c`.

For deatils on how to extend the system with new kernel ALisp functions implemented in C, storage management, etc., see document *doc/alisp.pdf*.

4.5 Documentation

Functions should be documented, supported "online" documentation is available only in for functions/variables available in ALisp.

Using the ALisp function `doc` one can get information on functions:

```
lisp 1> (doc 'createfunction)
"Create new OSQL function
resv, quant and pred = NIL => stored function
resv = FOREIGN => foreign function, resv = implementation name for table FF
otherwise derived function with select expression defined by resv, quant,
and pred"
```

Functions that can be accessed from within ALisp, can and should be documented. If you write the documentation in a special way, as described below, it is automatically extracted. It can then be viewed with the `doc` function.

ALisp functions can be written to contain a *Documentation-string*:

```
(defun fac (n)
  "Returns the factorial of N."
  (if (> n 0)
      (* n (fac (1- n)))
      1))
```

5 Extending the system

This section describes the common way of extending the system kernel with new features in C. There is a choice of implementing the extensions in existing files or add new files that implements the new features.

NOTICE: The description here is about extending the *kernel* of Amos II. If you are writing applications calling Amos II or being called from Amos II there is a much higher level API from C and Lisp documented in file *external.pdf*. The corresponding API for Java is documented in *javaapi.pdf*.

5.1 New files or use existing files

If the kernel feature added is an extension of an existing feature it is advised to use the file(s) with that feature. Example: If new AMOSQL functions implemented in ALisp are added it is quite natural to add the implementation to the file `amosfns.lsp` since that file does nothing but implements AMOSQL-functions.

If, however, the new feature is something completely new to the system the reasonable thing to do is to implement it in a new file or set of files.

5.2 Adding a new Lisp file

Add a new file containing source code written in ALisp is very easy. Write the file and give it a mnemoic name and extension `.lsp`. The location of the file should be amongst all other Lisp files i.e. `../AmosNT/lsp`. The file must then be read by the system at creation of the system. This is achieved by adding a load of the file in `init.lsp`.

5.3 Adding a new C file

Adding a new file containing source code written in C is a bit more hassle than adding a new file in Lisp.

Following has to be done

- 1: Implement the features in a file with extension `.c`

- 2: Create a file with extension `.h` containing interface declarations from the new file.
- 3: In file `amos.h` include the new `.h` file.
- 4: In the project makefile `system/Borland/amos.bpr` add the new `.c` file using Borland C++ builder.
- 5: Always use the predefined headers.

In step 2 above it is necessary to do as follows in order to catch any circular dependencies. Assume the new file is `aChack.c`. In `aChack.h` the following must be done:

```
#ifndef _aChack_h_
#define _aChack_h_
 <Declarations>
#endif
```

In step 4 above it is of importance to add the new `.c` file in a fitting group.

NOTICE: The length of a filename must **not** exceed 17 characters. This is due to the source file version package that does not function correctly if this restriction is violated. Still true???

5.4 Regression test

When the new features are working properly and the regression test runs well the regression test can be extended to test the new features. This is done by writing a new testfile with extension `.osql` located in `../AmosNT/regress`. The new regress file must then be called from `test.osql`. This is done by adding a line in `test.osql` as:

```
< 'mynewtest.osql';
```

6 Lisp programmer's Checklist

6.1 Symbols

The datatype *symbol* is documented in Sec. 3.1 in *Alisp User's Guide*. Read introduction carefully!

NOTICE that system hash table bound to variable `**SYMBOLHASHTABLE**` maps *print names* to symbols in order to make them unique. **NEVER** change `**SYMBOLHASHTABLE**`!

NOTICE that symbol print names in CommonLisp are always upper-cased when a symbol is created. For example:

```
(MKSYMBOL "aBc") -> symbol ABC
```

NOTICE that the obsolete function `MKATOM` is the same as `MKSYMBOL` but is NOT standard CommonLisp.

Symbols are often used in the system for upper-casing strings and for generating unique identifiers for strings. For example,

```
(EQ \"a\" \"a\") -> NIL                    (strings are not unique)
(EQ (MKSYMBOL \"a\")(MKSYMBOL \"a\")) -> T (symbols are unique)
(EQ (MKSYMBOL \"a\")(MKSYMBOL \"A\")) -> T (symbols are upper-case)
```

NOTICE that the cost of creating a symbol is *much* higher that creating a string. In particular it should be avoided to generate an unlimited number of symbols by calling `MKSYMBOL` on arbitrary strings.

NOTICE that symbols are *not* garbage collected and creating many of them causes storage leaks.

Variables in ObjectLog are represented using symbols. The function

```
(GENVAR)
```

is used whenever the system needs to generate a new variable. It generates variables `_V1, _V2, etc`. In order to avoid generating variables with higher and higher generation numbers (and thus unlimited number of symbols) the counter for the variable generation numbers is reset whenever a new function is compiled by the macro:

```
(RESETGENVAR FORM)
```

`RESETGENVAR` resets the variable generation number counter to 0, evaluates the form, and then restes the counter to its original value again.

7 C Programmer's Checklist

This section describes common programming errors, and warnings when adding C code to the kernel. More details can be found in *alisp.pdf*.

- Avoid using `C's assignment operator` for Amos II data structures (unless you know what you are doing). Always use `dcloid` and `a_setf` instead!
- The `unwind_protect_catch` code **must** be executed; never return directly out of the main code block. If `unwind_protect_end` is not executed after an exception, then the exception is not continued. Always execute `unwind_protect_end`, unless you really know what you are doing.

- Streams must be closed before saving image!
- The system may stop functioning if `breaks` are put on a system function that is part of the `break package` itself!
- The use of handles allows the storage manager to move the objects, making dereferenced handles dangling pointers. Thus dereferenced pointers may become incorrect once a system feature that cause data to move is called. Object allocation is the only system operation that may cause this. Thus, if a system function is called that is suspected to do object allocation (most do), the dereferencing MUST be redone. It is always safe to dereference through `dr` and to always use `dcloid(x)` and use `a_setf` for assignments.
- Never try to reset C arguments with `a_setf`; it will clobber the garbage collector.
- Don't forget to free the result of a Lisp function call (with `a_free`) in case it generates garbage.
- call_lisp automatically garbage collects its arguments upon return; thus temporary objects among the arguments are automatically freed. For example:

  ```
  call_lisp(mksymbol("prin1"),env,1,mkstring("hello world"));
  ```

  will allocate a new Lisp string, then print it, and then deallocate it on the exit from `call_lisp`.

  However, the automatic deallocation of temporary storage is NOT performed with direct C function calls. For example, with a direct C call to the C implementation of Lisp's `PRINT`, `printfn`, the above printing should become:

  ```
  dcloid(a1);
  ......
  a_setf(a1,mkstring("hello world"));
  printfn(env,a1);
  a_free(a1);
  ```

7.1 Parsing with Bison/Flex

The Bison/Flex parser generator is frequently used when parsing various languages. Bison/Flex can be run under Windows. Download the .exe files from `http://user.it.uu.se/~udbl/software/bison` and make sure that the PATH variable include their sources. The file `bison.simple` must be placed in `c:`
`tools`
`share`
`bison.simple`.

8 Making an Amos II Release

First, check out the latest version of Amos II from CVS (don't forget to check in everything You want to be part of the new release first) and move to the directory containing the newly checked out copy of the AMOS system. Standing in the `AmosNT/bin` directory, type `install all`. The system should now compile. Make sure that it passes the regression test.

The command procedure `mkrelease.bat` in `AmosNT` root directory makes a new release for export.

More here...

9 Bugs

If you should find any bugs in a released Amos II, then send a mail to `Tore.Risch@it.uu.se` describing the bug. If possible include an example detailed enough to recreate the bug. If Amos II dumps core you could run `Borland C++` and look at the stack to see in what function the fault occured.