Solutions of common problems

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

Table of Contents

This library contains the following modules:

`Message`	Error Reports (module deleted)
`Strings`	String Concatenation
`Counter`	Counting Symbol Occurrences
`MakeName`	Generating Optional Identifiers
`Sort`	Sorting Elements of an Array
`StrArith`	Character String Arithmetic

1 Error Reports

This module is NOT supported anymore.

Uses of the two macros `Message` and `MessageId` that were provided by this module have to be rewritten, since LIGA does NOT allow to hide accesses of `COORDREF`.

A macro call `Message(s, t)` can be rewritten `message(s, t, 0, COORDREF)`.

A macro call `MessageId(s, t, sym)` can be rewritten

```
message(s, CatStrInd(t, sym), 0, COORDREF)
```

where `CatStrInd` is a macro that stores the concatenation of two strings. It is provided by the module (see Chapter 2 [Strings], page 5):

```
$/Tech/Strings.specs
```

2 String Concatenation

This module provides functions for concatenation of strings, and the name `CharPtr` for the type `char *`.

The module is instantiated by

```
$/Tech/Strings.specs
```

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `Strings.h` is imported there.

The module defines the type name `CharPtr` for `char *`, especially to be used in LIDO specifications.

The module exports the functions

`CharPtr CatStrStr (CharPtr s1, CharPtr s2)`
: The strings `s1, s2` are concatenated. The resulting string is stored in the string memory of the `csm` module, and the pointer to it is returned.

`int IndCatStrStr (CharPtr s1, CharPtr s2)`
: The strings `s1, s2` are concatenated. The resulting string is stored in the string memory of the `csm` module, and its `StringTable` index is returned.

A macro `CatStrInd(s,i)` is exported it obtains the second of the concatenated strings by the index `i` into the `StringTable`. This macro is used to simplify composition of `message` texts.

3 Counting Symbol Occurrences

This module provides `.lido` computations that count all ocurrences of certain symbols within a certain subtree.

The module is instantiated by

```
    $/Tech/Counter.gnrc +instance=NAME :inst
```

The optional generic instance parameter `NAME` identifies the particular instance.

The module provides the following computational roles:

`NAMECount` is associated to the grammar symbols that shall be counted. `NAMECount.NAMECount` yields the occurrence number in the tree.

`NAMERootCount` specifies the subtree containing the `NAMECount` occurrences. The number of occurrences found in the subtree can be obtained by the attribute `NAMERootCount.NAMECountResult`.

The default is that counting starts from 1 and is incremented by 1. The start value can be adjusted by overriding the computation `NAMERootCount.NAMEInitCount = 0;` with a computation of a suitable value. The increment can be adjusted by overriding the computation of `NAMECount.NAMEIncrement`.

|NAME|RootCount is inherited by the grammar root by default.

`NAMERootCount` can be associated to recursive grammar symbols. Any symbol that has `NAMECount` associated must belong to a subtree such that `NAMERootCount` is associated to its root.

4 Generating Optional Identifiers

This C module implements functions that turn their arguments into strings which then play the role of identifiers as if they occurred in the input, i.e. they are entered in the identifier table, and their symbol code is returned.

The module is instantiated by

```
        $/Tech/MakeName.gnrc +instance=IDENT :inst
```

The generic instance parameter `IDENT` has to be set to the terminal symbol used for identifiers.

The module can also be used to generate and store identifiers in processors that do not have a scanner and a parser. In this case the `+instance` parameter has to be omitted.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `MakeName.h` is imported there.

The module exports the following functions:

`int MakeName (char *c)`
: `c` is a character string that may coincide with an already existing name. The result the is the encoding of the name `c`.

`int GenerateName (char *c)`
: `c` is a character string. The name is generated by appending a number to the prefix `c` such that the name is different from all others encountered so far (on input and generated).

`int IdnNumb (int id, int num)`
: `id` is the encoding of an existing identifier; `num` is a nonnegative number. The new name is formed by appending the number to the identifier string. It may coincide with an already existing name. This function may be used to derive arbitrary names from existing ones.

`int PreIdnPost (char *pre, int id, char *post)`
: `id` is the encoding of an existing identifier, `pre` and `post` are arbitrary character strings. The generated name is formed by catenation of `pre`, the identifier string and `post`. It may coincide with an already existing name. This function may be used to derive different names (e. g. for different target objects) from an existing name.

5 Computing a Hash Value

This C module computes a 32-bit hash of the contents of specified memory. Every bit of the memory contents affects every bit of the hash. The probability that the same hash will be computed for different memory contents is very low.

The module is instantiated by

```
$/Tech/Hash.specs
```

All entities exported by this module can be used in specifications of type `.lido`, `.init`, and `.finl`. They can also be used in C modules if the interface file `hash.h` is imported there.

The module exports the following entities:

`ub1`
: A typedef identifier defining an unsigned byte. The memory over which a hash is to be computed is made up of a set of contiguous blocks of `ub1` values.

`ub4`
: A typedef identifier defining an unsigned 32-bit value. The hash computed is a single `ub4` value.

`ub4 hash(ub1 *block, size_t length, ub4 previous)`
: A function computing a hash value. The '`block`' argument is a pointer to a contiguous sequence of '`length`' unsigned bytes; '`previous`' is the hash computed from other contiguous sequences in the specified memory area. (If '`block`' is the first sequence of the area, '`previous`' should be '`0`'.) The result of `hash` is a hash of all of the contiguous sequences considered so far.

In the simplest case, we need to compute a hash for the contents of a single contiguous block of memory. For example, here is a call to compute a hash of a single string pointed to by '`str`':

```
hash(str, strlen(str), 0)
```

A more complex situation is when there are several related areas of memory, and we need to compute a single hash for all of them. Suppose that there was a pair of strings, pointed to by '`str1`' and '`str2`', which constituted a conceptual unit. Here are two ways to compute a single hash for the pair:

```
hash(str1, strlen(str1), hash(str2, strlen(str2), 0))
hash(str2, strlen(str2), hash(str1, strlen(str1), 0))
```

Either of these sequences would be perfectly satisfactory, but the resulting values would differ.

When computing the hash of an array or structure, it is important to realize that there may be padding with unknown content involved. Consider the following variable declaration:

```
struct{ char c; int i; } foo, bar;
```

On some machines, the compiler may insert three bytes of padding between the end of field '`c`' and the beginning of field '`i`'. There is no guarantee that these three bytes will be initialized in a particular way. Thus '`hash(foo, sizeof(foo), 0)`' and '`hash(bar, sizeof(bar), 0)`' may not yield the same result when '`foo`' and '`bar`' have identical field values; the hash also depends on the contents of the padding. Unless you know that there is no padding present, or that padding is always intialized in the same way, the only safe approach is to hash the fields as separate memory areas:

```
hash (foo.c, sizeof(foo.c), hash(foo.i, sizeof(foo.i), 0))
```

6 Sorting Elements of an Array

This C module implements a function that sorts the elements of an array in place. It uses the O(n log n) Heapsort algorithm.

The module is instantiated by

```
$/Tech/Sort.gnrc +instance=TYPE +referto=HDR :inst
```

where `TYPE` is the name of the array element type and `HDR.h` is a file that defines the array element type, e.g.

```
$/Adt/Sort.gnrc+instance=DefTableKey +referto=deftbl :inst
```

If the element type is predefined in C the `referto` parameter is omitted, e.g.

```
$/Adt/Sort.gnrc+instance=int :inst
```

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `SortTYPE.h` is imported there.

The module exports the following function:

`void SortTYPE (TYPE *arr, size_t n, TYPECmpFctType cmp)`

`arr` points to the array to be sorted, which contains `n` elements. `cmp` is the function that defines the collating sequence for the elements. It's signature is `TYPE, TYPE -> int`; it yields -1 if the left argument should precede the right in the sorted array, +1 if the left argument should follow the right in the sorted array, and 0 if the order of the elements in the sorted array is immaterial.

7 Character String Arithmetic

StrArith is a wrapper for the `strmath` library routines (see Section "Character String Arithmetic" in *Library Reference Manual*). It is instantiated by

```
$/Tech/StrArith.gnrc +instance=NAME +referto=BASE :inst
```

where `NAME` is a prefix for the operator names and `BASE` is the radix of the numbers to be manipulated. If the `instance` parameter is omitted then the operator names have an empty prefix; if the `referto` parameter is omitted then the radix is 10.

All operations exported by this module can be used in specifications of type '.lido', '.init', and '.finl'. They can also be used in C modules if the interface file 'NAMEStrArith.h' is imported there.

Strings representing numbers are all stored in the string table. Their form is common in programming languages:

```
[+/-][d*][.][d*][e[+/-]d*]
```

Here `[]` indicate optional parts, `+/-` indicates that a sign may be present, `d` indicates digits in the chosen radix, and `*` indicates repetition. The optional dot separates the integer and fractional parts of a number, and `e` stands for an exponent marker. The actual characters used to represent digits, signs, fractional separators and exponent symbols are determined by default or by settings established by the `strmath` operation (see Section "Character String Arithmetic" in *Library Reference Manual*).

An instantiation of `StrArith` with instance parameter `NAME` provides the following operations:

`int NAMEStrAdd(int left, int right)`
`int NAMEStrSub(int left, int right)`
`int NAMEStrMul(int left, int right)`
`int NAMEStrDiv(int left, int right)`
`int NAMEStrQuo(int left, int right)`
`int NAMEStrRem(int left, int right)`
`int NAMEStrExp(int left, int right)`

Dyadic arithmetic operations. The two arguments are string table indices representing the left and right operands. The result is the string table index of a string representing the result.

`NAMEStrDiv` is a real division, possibly yielding a result with a fractional part. `NAMEStrQuo` yields the integer quotient from the division and `NAMEStrRem` yields the integer remainder from the division. `NAMEStrExp` raises the first operand to the power given by the second operand.

`int NAMEStrNeg(int opnd)`
`int NAMEStrSqrt(int opnd)`

Monadic arithmetic operations. The argument is a string table index representing the operand. The result is the string table index of a string representing the result.

`int NAMEStrNorm(int opnd, int oldradix, int newradix, char *symbs)`

Normalizes a value, performing radix conversion if necessary. The first argument is a string table index representing the operand. The second and third arguments are the radix values for the conversion. The result is the string table index of a string representing the result. Its format depends on the content of the fourth argument:

`symbs=0`
: Whole number and fraction parts separated by the defined fractional separator unless the result can be expressed as an integral value, exponent marker and exponent if the length would exceed `integer_size` digits.

`symbs=""`
: Sequence of digits if the length does not exceed `integer_size` digits, otherwise the empty string (which is represented by string table index 0).

`symbs=non-empty quoted string`
: A fractional separator is guaranteed to appear in the result. The first character of the string is taken as the exponent marker. If there are additional characters in the string, then they will be taken as the fractional separator, the minus sign, and the plus sign respectively. (The characters normally defined for these purposes will be used if the corresponding character does not appear in the string.)

Here is a fragment of a specification for a calculator that uses `StrArith` to implement the arithmetic:

```
ATTR val: int;

RULE: Program ::= Expr COMPUTE
  printf("%s\n", StringTable(Expr.val));
END;

RULE: Expr ::= Expr '+' Expr COMPUTE
  Expr[1].val=StrAdd(Expr[2].val,Expr[3].val);
END;

...

RULE: Expr ::= '-' Expr COMPUTE
  Expr[1].val=StrNeg(Expr[2].val);
END;

RULE: Expr ::= 'sqrt' '(' Expr ')' COMPUTE
  Expr[1].val=StrSqrt(Expr[2].val);
END;

RULE: Expr ::= Constant COMPUTE
  Expr.val=Constant;
END;
```

`StrArith` was instantiated without parameters for this example:

```
$/Tech/StrArith.gnrc :inst
```

Index