Gamma Programming Language Declarations

Contents

Introduction

The Gamma programming language is a *source language* in which to write *source code*. A *translator* may then *translate* source code written in the language into *object code* written in an *object language*.

This document specifies the namespace units and declarations therein provided to programmers by translators.

Interpretation

Except where explicitly stated otherwise, this document has a prescriptive intent: a translator will behave so that all statements in this document hold.

Conventions

Except where explicitly stated, text in `this font` should be valid Gamma source code.

1 Environment

The environment namespace, `env`, declare types, datums and subprograms for interaction with the program's environment, such as dynamic memory allocation and input/output.

1.1 Memory

The memory namespace, `mem`, declares subprograms for handling memory.

`sym setbuf: (b: {null: nsize; val: (@byte)[]}, m: nsize)`
If `m` is not zero, sets `b.val` to the address of an array of size `m`, otherwise sets `b.null` to `.b`.

1.2 Time

The time namespace, `time`, declares types, datums and subprograms for handling calculations over time values.

1.2.1 Types

`time`
A type whose values represent moments in time.

`clock`
A type whose values represent devices emitting `time` values.

1.2.2 Datums

1.2.3 Subprograms

1.3 Localization

The localization namespace, `loc`, declares types, datums and subprograms for interpreting and producing data according to a locale.

1.3.1 Types

The following types are defined in every module binding against the localization namespace:

- `string`, the *string* type; and
- `grapheme`, the *grapheme* type.

Each subprogram is declared within namespaces whose names form identifiers of the form
*locale**codeset*[*modifier*]
where *locale* is the ISO 15897 name of the locale, *codeset* is the name of the encoding, and *modifier* is an optional modifier which further qualifies the locale.

`sym nmax: nmax(s: string, b: n1),`
Subprogram interpreting `s` as a natural integer to the base `b`.

`sym zmax: zmax(s: string, b: n1),`
Subprogram interpreting `s` as a relative integer to the base `b`.

`sym dmax: dmax(s: string),`
Subprogram interpreting `s` as a single-precision real.

`sym time: time(s: string),`
Subprogram interpreting `s` as a time.

`sym string: string(n: nmax, b: n1),`
Subprogram representing `n` as a string to the base `b`.

`sym string: string(z: zmax, b: n1),`
Subprogram representing `z` as a string to the base `b`.

`sym string: string(r: dmax),`
Subprogram representing `r` as a string.

`sym string: string(t: time),`
Subprogram representing `t` as a string.

`sym coll: zmax(l: string, r: string),`
Subprogram collating `l` and `r`.
The result is:

- negative if `l` precedes `r`,
- zero if `l` is equivalent to `r`, and
- positive if `l` succeeds `r`.

The absolute value of the result indicates the index after where the first difference occurs.
For example, `is_is\utf8\coll("æ","ae") == 1` and `en_us\utf8\coll("æ","ae") == 0`.

`sym enc: string(b: byte[]),`
Subprogram transforming `b` into a string.

`sym dec: byte(s: string)[],`
Subprogram transforming `s` into a byte array.

`sym slen: nsize(s: string),`
Subprogram returning the length of `s` interpreted as a sequence of graphemes.
For example, `is_is\utf8\slen("æ") == 1` and `en_us\utf8\slen("æ") == 2`.

`sym glen: nsize(g: grapheme)`
Subprogram returning the length of the encoding of `g` according to the current locale.

1.4 Input/Output

The input/output namespace, `io`, declares types and subprograms for interacting with external data sources and targets.

1.4.1 Types

Translators define the following types:

`dev`
A type whose values describe distinct devices.

`pos`
A type whose values designate different positions in device storage.

1.4.2 Subprograms

`sym tell: err(p: pos, d: dev)`
Sets `p` to the position currently associated with `d`.

`sym seek: err(d: dev, p: pos)`
Sets the position currently associated with `d` to `p`.

`sym stream: err(tgt: byte[][], src: dev)`
Reads from `src` the amount of data which `tgt` can contain.

`sym stream: err(tgt: dev, src: byte[][])`
Writes to `tgt` the amount of data in `src`.

`sym block: err(tgt: byte[][], src: dev, p: pos)`
Reads from `src` at `pos` the amount of data which `tgt` can contain.

`sym block: err(tgt: dev, src: byte[][], p: pos)`
Writes to `tgt` at `pos` the amount of data in `src`.

1.5 Threads

The thread namespace, `thread`, defines types, datums and subprograms for starting concurrent executions of a module.

1.5.1 Types

`thread`
A type whose values refer to threads of a module.

1.5.2 Data

`sym self: thread`
Variable whose value refers to the thread accessing it.

1.5.3 Subprograms

`sym join: (t: thread)`
Waits for `t` to terminate.

`sym fork: (t: thread, a: {null: nsize; val: (@byte)[]})`
Starts `t` with either no arguments if `a.null == .a`, or otherwise the arguments pointed to by `a.val`.

1.6 Assembly

The assembly namespace, `asm`, defines subprograms which provide access to the underlying object language facilities in a translator-agnostic manner.

This section is descriptive; the definitions should be provided by an affiliated or associated organization or individual. An example will be given to illustrate.

1.6.1 RISC-V RV32I

The namespace for RV32I, `rv\32`, defines datums and subprograms for controlling RV32I object code produced by the translator.

1.6.1.1 Types

`reg`
Type containing the values which can be stored in a general purpose register.

`facs`
Type containing whose values enumerate the access kinds controlled by the `FENCE` instruction (see below).

1.6.1.2 Datums

```
sym x0:  reg!,
    x1:  reg,
    x2:  reg,
   \* ... *\
    x31: reg,
    pc:  reg
```

Symbols providing access to the values in the registers with the corresponding names.

1.6.1.3 Subprograms

```
sym lw:  (rd: reg?, rs1: reg, imm: n2), sw:  (rs2: reg!, rs1: reg, imm: n2),
    lhu: (rd: reg?, rs1: reg, imm: n2),
    lh:  (rd: reg?, rs1: reg, imm: n2), sh:  (rs2: reg!, rs1: reg, imm: n2),
    lbu: (rd: reg?, rs1: reg, imm: n2),
    lb:  (rd: reg?, rs1: reg, imm: n2), sb:  (rs2: reg!, rs1: reg, imm: n2)
```

Subprograms performing the load and store instructions, and variants.

```
sym add:  (rd: reg?, rs1: reg, rs2: reg),
    addi: (rd: reg?, rs1: reg, imm: n4!)
```

Subprograms performing the `ADD` instructions and variants.

```
sym nop:  ()
```

Subprogram performing the `NOP` pseudoinstruction.

```
sym fence: (pr: facs, su: facs)
```

Subprogram performing the `FENCE` instruction.

All other instructions are defined analogously.

1.7 Operating system

The operating system namespace, `os`, contains namespaces defining subprograms for interfacing directly with the operating system.

This section is descriptive; the definitions should be provided by an affiliated or associated organization or individual. An example would be a POSIX namespace, `posix`, containing types, programs and subprograms for interfacing with POSIX compliant operating systems, such as AIX and QNX.

2 Language

The language namespace, `lang`, declares subprograms and datums for tasks involving types with formal operations, as well as information regarding the translator. Examples are complex arithmetic and querying the limits of built-in types such as `zmax`.

2.1 Translator Information

The translator information, `tran`, gives the code access to information on translator-defined details. Examples are the size of individual cells in memory and limits of provided types.

2.1.1 Type Properties

The type properties provided are for types defined as part of the language; types defined as part of the standard declarations will have their properties provided with their definitions.

2.1.1.1 Fixed-Point Types

`sym N1_MAX: n1,`
Maximum value of the type `n1`.

`N2_MAX: n2,`
Maximum value of the type `n2`.

`N4_MAX: n4,`
Maximum value of the type `n4`.

`N8_MAX: n8,`
Maximum value of the type `n8`.

`N16_MAX: n16,`
Maximum value of the type `n16`.

`N32_MAX: n32,`
Maximum value of the type `n32`.

`N_MAX: nmax`
Maximum value of the type `nmax`, and thus the absolute maximum representable value.

`sym Z1_MIN: z1,`
Minimum value of the type `z1`.

`Z2_MIN: z2,`
Minimum value of the type `z2`.

`Z4_MIN: z4,`
Minimum value of the type `z4`.

`Z8_MIN: z8,`
Minimum value of the type `z8`.

`Z16_MIN: z16,`
Minimum value of the type `z16`.

`Z32_MIN: z32,`
Minimum value of the type `z32`.

`ZMIN: zmin`
Minimum value of the type `zmax`, and thus the absolute minimum representable value.

`sym Z1_MAX: z1,`
Maximum value of the type `z1`.

`Z2_MAX: z2,`
Maximum value of the type `z2`.

`Z4_MAX: z4,`
Maximum value of the type `z4`.

`Z8_MAX: z8,`
Maximum value of the type `z8`.

`Z16_MAX: z16,`
Maximum value of the type `z16`.

`Z32_MAX: z32`
Maximum value of the type `z32`.

`ZMAX: zmax`
Maximum value of the type `zmax`, and thus the maximum representable relative value.

2.1.1.2 Floating-Point Types

`sym D1_MIN: d1,`
Minimum value of the type `d1`.

`D1_MIN_U: d1,`
Minimum unnormalized value of the type `d1`.

`D1_MAX: d1,`
Maximum value of the type `d1`.

`E_RADIX: n1,`
Radix of the exponent.

`D1_EPSILON: d1,`
Minimum epsilon of the type `d1`.

`D1_E_MIN: d1,`
Minimum exponent value of the type `d1`.

`D1_E_MAX: d1`
Maximum exponent value of the type `d1`.

2.2 Types

The types namespace, `types`, defines types for assisting in various data processing tasks. Examples include complex numbers and containers.

2.2.1 Complex Types

The complex type namespace, `complex`, defines the enumerated types `c1`, `c2`, `c4` and `cmax`, as well as the imaginary unit literal, `i`.

Translators may define the additional complex types `c8`, `c16` and `c32`.

The complex types' sizes are in a geometric progression of common ratio 2; their values and properties are translator-defined.

The following operations yield values of complex type, ordered by precedence:

1. – `+:` *`ctype`*`(`*`ctype`*`)`, *unary plus*;
   – `-:` *`ctype`*`(`*`ctype`*`)`, *unary minus*;
2. `/:` *`ctype`*`(`*`ktype1`*`,`*`ktype2`*`)`, *division*;
3. `*:` *`ctype`*`(`*`ktype1`*`,`*`ktype2`*`)`, *multiplication*;
4. – `+:` *`ctype`*`(`*`ktype1`*`,`*`ktype2`*`)`, *addition*;
   – `-:` *`ctype`*`(`*`ktype1`*`,`*`ktype2`*`)`, *subtraction*;

where *`ctype`* is any of the above complex types, and either:

- *`ktype1`* is complex and *`ktype2`* is not complex; or
- *`ktype2`* is complex and *`ktype1`* is not complex.

2.2.1.1 Subprograms

Except where explicitly stated, the subprograms defined here are provided in overloaded versions for performance reasons.

`sym real: dmax(c: cmax)`
Returns the real part of `c`.

`sym imag: dmax(c: cmax)`
Returns the real part of `c`.

`sym arg:  dmax(c: cmax)`
Returns the argument of `c`.

`sym abs:  dmax(c: cmax)`
Returns the modulus (absolute value) of `c`.

2.2.2 Containers

The containers namespace, `cont`, declares types, datums and subprograms which provide common abstract data types, such as queues and maps.

2.2.2.1 Deques

A *deque* is a container permitting ordered retrieval of elements stored within. The deques presented here in particular are priority queues, meaning elements are retrieved in decreasing priority.

`sym deque: deque(d: (@byte[])[], s: (a: byte[], o: order))`
Constructs a deque initialized to `d` and sorted by `s`.

`sym deque: deque(d: (@byte[])[])`
Constructs a deque initialized to `d` and sorted by the standard sort.

`sym insert: (t: deque, e: (@byte)[], n: nsize)`
Adds `@e` to `t` with priority `n`.

`sym remove: (e: (@byte)[], t: deque)`
Removes `e` from `t`.

`sym max: (@byte(t: deque))[]`
Returns a pointer to the maximum of `t`.

`sym max: nsize(t: deque)`
Returns the highest priority of all in `t`.

`sym min: (@byte(t: deque))[]`
Returns a pointer to the minimum of `t`.

`sym min: nsize(t: deque)`
Returns the lowest priority of all in `t`.

2.2.2.2 Maps

A *map* is a container associating *keys* to *values* in a functional manner, i.e. $i = j \Rightarrow M[i] = M[j]$.

`sym map: map(d: {k: key, v: (@byte)[]}[])`
Creates a map from the pairs in `d`.

`sym insert: (m: map, k: key, v: (@byte)[])`
Adds (`k`, `v`) to `m`.

`sym remove: (k: key, m: map)`
Removes `k` and its associated value from `m`.

`sym get: (v: (@byte)[], m: map, k: key)`
Sets `v` to the value of `m` for `k`.

`sym set: (m: map, k: key, v: (@byte)[])`
Sets the value of `m` for `k` to `v`.

`sym has: bool(m: map, k: key)`
Returns whether `k` is in the domain of `m`.

`sym has: bool(m: map, v: (@byte)[])`
Returns whether `v` is in the range of `m`.

2.2.2.3 Sets

A *set* is a container which contains those values satisfying a predicate in an extensional manner.

`sym set: set(v: (@byte[])[])`
Creates a set from the values in `v`.

`sym insert: (s: set, v: (@byte)[])`
Adds `v` to `s`.

`sym remove: (v: (@byte)[], s: set)`
Removes `v` from `s`.

`sym has: bool(s: set, v: (@byte)[])`
Returns whether `v` is in `s`.

3 Algorithmics

The algorithmics namespace, `algo`, declares types, datums and subprograms for performing tasks not requiring interaction with the environment, such as data processing and mathematical processes.

3.1 Buffers

Buffers, for the purposes of this document, are values of type `byte[]`.

`sym fiss:   byte(s: byte[], d: byte[])[][]`
Returns an array of subsequences of `s` separated by `d`.

`sym fuse:   byte(s: byte[][], j: byte[])[]`
Returns an array composed of the subsequences of `s` joined by `j`.

`sym search: nsize(s: byte[], p: byte[])[]`
Returns an array of indices into `s` at which the sequence `p` occurs.

`sym set:  err(a: byte[], v: byte)`
Sets all elements in `a` to the value of `v`.

`sym move: err(d: byte[], s: byte![])`
Sets all elements in `d` to the values of the elements in `s`.

`sym copy: err(d: byte[], s: byte![])`
Sets all elements in `d` to the values of the elements in `s`. `d` and `s` do not overlap.

3.2 Functions

The functions namespace, `fun`, declares mainly subprograms related to common algorithms such as hashes and probabilities.

3.2.1 Hashes

`sym hash: nsize(a: byte[])`
Calculates the hash value of `a`.

3.2.2 Probabilities

Except where explicitly stated, all subprograms declared here have an overload taking the `rneng` to use as the first parameter.

`sym multinomial: d4(t: n4[], p: d4[])`
Returns the probability of a trial outcome sequence `t` given the distribution `p`.

`sym categorical: n4(p: d4[])`
Returns the outcome of a categorical trial given the distribution `p`.

`sym uniform_z: z4(a: z4, b: z4)`
Returns an integer chosen uniformly between `a` and `b`.

`sym uniform_d: d4(a: d4, b: d4)`
Returns a decimal chosen uniformly between `a` and `b`.

`sym poisson: d4(k: d4, p: d4)`
Returns the probability of `k` events according to a poisson distribution with mean rate `p`.

`sym exponen: d4(t: d4, p: d4)`
Returns the probability of an event in the interval `t` according to an exponential distribution with rate `p`.

3.2.3 Sorts

`sym shuf: (a: byte[], g: rneng)`
Shuffles `a` using `g` as the randomizing engine.

`sym sort: (a: byte[], o: order)`
Sorts `a` using `o` as the comparison function.

3.3 Mathematics

The mathematics namespace, `math`, contains types, datums and subprograms for common mathematical work.

3.3.1 Graphs

`sym source: node(e: edge)`
Returns the node from which `e` leaves.

`sym target: node(e: edge)`
Returns the node to which `e` goes.

`sym value: (v: byte[], n: node)`
Sets `v` to the value of `n`.

`sym value: (n: node, v: byte[])`
Sets the value of `n` to `v`.

`sym value: (v: byte[], e: edge)`
Sets `v` to the value of `e`.

`sym value: (e: edge, v: byte[])`
Sets the value of `e` to `v`.

`sym node: (n: node, g: graph)`
Removes `n` from `g`.

`sym node: (g: graph, n: node)`
Adds `n` to `g`.

`sym edge: (e: edge, g: graph)`
Removes `e` from `g`.

`sym edge: (g: graph, e: edge)`
Adds `e` to `g`.

3.3.2 Subprograms

Except where explicitly stated, the subprograms are provided in overloaded versions for performance reasons.

`sym sin: dmax(a: dmax)`
Calculates the sine of `a`.

`sym cos: dmax(a: dmax)`
Calculates the cosine of `a`.

`sym tan: dmax(a: dmax)`
Calculates the tangent of `a`.

`sym asin: dmax(a: dmax)`
Calculates the arcsine of `a`. If `a` is not between -1 and 1 inclusive, the behavior is translator-defined.

`sym acos: dmax(a: dmax)`
Calculates the arccosine of `a`. If `a` is not between -1 and 1 inclusive, the behavior is translator-defined.

`sym atan: dmax(a: dmax)`
Calculates the arctangent of `a`.

`sym ln: dmax(a: dmax)`
Calculates the natural logarithm of `a`. If `a` is negative, the behavior is translator-defined.