Functorial Methods applied in the

π-Nets Programming Language

Methods to arrive at a

Simple Programming Language

Fritz Mayer-Lindenberg, TUHH

What is a programming language made for?

+ for formally describing composite computations ('algorithms')

  such that the description can be automatically processed to control the execution of the operations on a computing machine.

  (our computers are built to carry out individual arithmetic operations in a few ns such that input, output, an storage of data must be automated).

- formal languages are used in mathematical logic and sometimes serve for automatic theorem proving. Programs are sometimes 'verified' or tried to be proved to be correct. For the execution on a machine only input data and some intermediate data may need to be checked to meet certain conditions.

- Programming languages can also be used to just specify some desired processing, or to define the behavior of external input and output and optionally derive a simulation of an embedded system.

Notion of 'Algorithms' 'FNA'

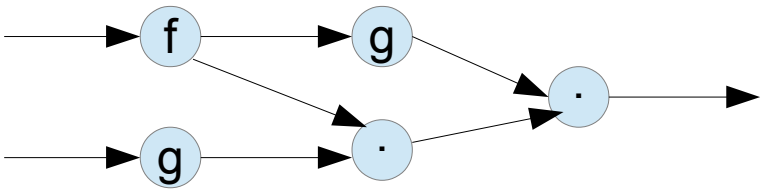
- Algorithms are given by a composition scheme (a directed graph S) with a set C of compute nodes and an assignment $\eta: C \rightarrow O$ of nodes to types of computational operations with compatible signatures. The composition scheme involves

  compositions of functions and operations (f°g)
  branches to select between (the results of) alternative sub computations

  Algorithms not performing memory access operations but only numeric operations/functions compute pure functions mapping numbers to numbers.

  .. are supposed to do so after a finite number of selected steps.

  Operations to be composed are the operations specified for the available data types.



- Algorithms will suffer from

  errors due to applying operations/functions to data outside their domains

  approximation errors due to the finiteness requirement

  interpolation errors once functions are represented by finite tables only

  rounding/overflow errors depending on the number codes used on a machine

Requirements on the π–Nets Programming Language

intended for numerical applications (signal processing, robotics etc.),

particularly embedded FPGA/SoC applications, HPC

- support simple FPGA based processors, small memories, no caches
- support numbers represented by various codes including non-standard ones
- support heterogeneous networks of FPGA based processors, distrib. memory
- define and support heterogeneous networks of FPGA/SoC/processor nodes
- provide basic real time control for IO operations

Extra requirement: .. besides supporting readable and well-structured programs

- **do so in a simple way to arrive at a small yet powerful programming tool**

..see SoC documentation for special motivation

The Fifth programming language (1985-1995) .. proving simplicity is possible

- supporting several mirocontrollers/-processors and small networks of such through preconfigured versions of the compiler, integrated assemblers
  Z80, 6301, 6809, 8031, 8086, 68000, 80166, 78312, Transputer, 2181, Sharc

- machine oriented providing bit field operations
  HW abstraction by programming for a stack oriented virtual machine

- infix notation for numerical expressions, number code input (fixed/float)

- multiple threads, process local variables

- small integrated programming, compiler and debugging environment
  implemented in Fifth and compiled for the PC
  interactive testing by linking the execution environment to the PC

- successfully used in a number of industry projects

Could some subset of the mathematical notation in textbooks be part of a p.language?

* Some nice notational features:

- values defined by expressions don't need a termination symbol
- names for functions/data can freely use special symbols, Greek characters
- can use subscripts and superscripts, can chain comparisons
- formulas can extend above and below a baseline (integrals, fractions)
- conditional, even recursive assignments extending over several lines

* Mixing with nat.lang. w/o extra indications

- formulas often embedded into nat.lang. Sentences
- math.texts are edited for the human reader, headers, comments
- textual references to previous definitions/contexts

Notions lacking a mathematical counterpart

- IO, communications, reading and writing variables, processes/threads, real time

Mathematical constructors w/o a counterpart on a computer

- defining infinite sets such as cosets of a vector sub space or infinite sub group
- defining sets of equivalence classes
- using images of some infinit set under a mapping
- using existence results proved non-constructively

Algorithms and proofs

- algorithms can be extracted from constructive proofs which yet focus on verification

Methods to achieve simplicity

- restrict range of applications and targets

  πN: numerical apps, simple target procs, static allocation

  choice of paradigms to identify/restrict programming task

- abstraction from implementation techniques

- small number of algorithmic constructors and a simple syntax

- small number of data types and type constructors

  πN: distinguish data types by their operations only

- supply powerful predefined operations

  πN: tuple operations (e.g. vector operations '+','-' w/o loops)

- **apply functorial methods as high-level compile-time functions**

  overloading symbols (names of operations)

- support parallelism, real time, target definition etc. in simple ways

  using hierarchic definitions to cope with complex apps

- provide a simple programming amd debugging environment

Data operated on by π–Nets programs:

fixed size tuples of real* numbers (finite fixed size tuple valued function tables)

tuple/table entries may be 'invalid'

.. no 'row' or 'column' vectors

no other predefined or derived data sets

predefined tuple operations:

apply tuple as a function, composition

linear/multilinear vector operations

(using ideal real arithmetics)

polynomial operations

interpolation operations

.. important basic selection ..

set operations

(extend by libraries of algs)

relations

.. used to test&branch

Algorithms are statically defined, pure program functions:

map tuples to tuples

.. compatible with tuples applied as functions

may have invalid results

.. can be tested for to branch

always terminate

.. to be vf'd by the programmer

.. maybe, with an invalid result

.. recursion supported to limited depth only

program functions can be grouped into 'optype' definitions and then redefine/ overload predefined op symbols such as '+' and '*'; the optype names are used to select the definition to be applied (simple substitute for data type definitions)

* could use rational/algebraic/computable numbers instead, encode two numbers (scaling, error)

Tuples can represent various mathematical objects as 'coordinates' dep. on choices

- tuple coordinates are the same for vectors, linear forms, and tensors etc.
  the distinction is through the applied operations, and their index mapping

- vector/tensor coordinates depend on a choice of basis
  algorithms may need to keep track of the basis and to perform base changes
  the handling of units is a special case

- a small number of alternative coordinates (e.g. charts of a manifold) can be handled using a dynamic index parameter and indexed functions selected by it

- mathematical objects defined as possibly infinite equivalence classes must be handled by using coordinates for representatives of the classes. The classes may require changes of their representative. In easy cases they don't, e.g. modulo classes of numbers or polynomials, or correspond to base changes.

  Example: associating a vector bundle with fiber V to a principal G-bundle P … $P \times V/\sim$

  Rem: unordered finite sets of numbers can be represented as equivalence classes of tuples

Details on the π–Nets tuple operations and expressions: .. indexes are 0…n-1

.. expressions bound by infix ops with a result

applying a unary function/operation	f x	g(x,y)	f apply-to x
use nested function calls within expressions	h f x		
applying a binary (2-argument) operation	x + y·z		x apply-add ..
applying a tuple as a linear function	y x	A x	
nested applications are right associative	f y x	y x z	
applying a tuple as a function, compose tuples	y.i	y:k.i	y ° x
applying a tuple y as a polynomial	y pol x		
applying a tuple y by interpolating from it	y ipl x		

.. functions and data tuples distinguished by a symbol attribute

.. results of expressions can be named for further references

relations w/o results can be 'chained', too $0 \leq x \leq 1$

A non-standard tupel operation:

$V = IR^k$ vector space, dim V = k, base $e_0,..,e_{k-1}$

$\Lambda^m V = IR^s$ outer product space, dim $\Lambda^m V = s$ base $e_0/\backslash...../\backslash e_{m-1}$, …

$/\backslash: \Lambda^m V \times V \rightarrow \Lambda^{m+1} V$ outer product, IR^s $\quad IR^k \times IR^s \rightarrow IR^s$

$V \times \Lambda^m V \rightarrow \Lambda^{m+1} V$ $\quad IR^s \times IR^k \rightarrow IR^t$

Application 1: determine whether a vector is in a given subspace $W \subset V$

Application 2: compute the determinant of k vectors

Application 3: a tupel x accessed as x:s is a sampled differential form

Question to the language designer:

Do the presumed applications legitimate the inclusion of /\
as a predefined tuple operation ??

.. more

- tuple literals are lists of numbers x, y, z, … , w
  or evaluations of some function/alg f on 0..n-1 n:f

- functions/operations map k tuples to a single tuple

- two-dimensional indexing reads x:m.i.j or x:m.(i,j)
  and extends to more dimensions x:(m,n).(i,j,k) (a number)
  x:(m,n).(i,j) (an n-tuple)

- separate operators '°', '.' → simple vector syntax and multidim. indexes

.. compatibility of algorithmically defined functions and tuples

- tuples of functions/algs can be formed, too f, g, h
  define another function, application (f,g) x = (f(x),g(x)) . (f,g)(x,y)=(f(x),g(y))

  .. compatible with matrix*vector notation A x or A:n x

- define compositions of functions w/o arg.refs. f°g (f°g) x = f g x = f (g x)

Functors vs functorial substitution

In category theory, *functors* map objects of one category to objects of another, e.g. a vector space V to its dual V*, and morphisms between objects to morphisms between the corresponding objects in a way compatible with composition, e.g. a linear map $h:V\rightarrow W$ between vector spaces to the transposed map $h^T:W^*\rightarrow V^*$.

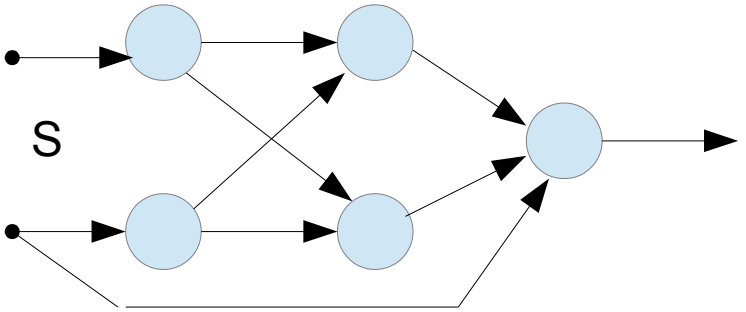
An algorithm specifying multiple compositions (a finite number) is then mapped to a similar algorithm in the target category, the 'algorithm' $f\circ g$ e.g. to the algorithm $g^T\circ f^T$.

In this presentation, *functorial substitution* refers to a method applying to algorithms (defined in a programming language), namely the formal substitution of their operations by associated ones operating on other data yet maintaining the composition scheme. The substitution also applies to operations/functions with multiple arguments and to conditions, branches and recursions in order to cover all algorithmic structures.

Functorial substitution can also be applied if it is defined for the function computed by an algorithm, too (not only predefined ones) and is not a functor. Then for the function F defined by an algorithm A, the substituted algorithm A' need not be an algorithm for the substitute F'. If F is 'called' in another algorithm, there is the choice to either substitute F by F' or to substitute it by the function computed by A'.

The substitution may be applied to predefined functions by choosing algorithms for them. It may be an automatic compiler operation or be on explicit commands yet w/o needing to explicitly redefine the composition scheme, thus simplifying programming.

Functorial substitution

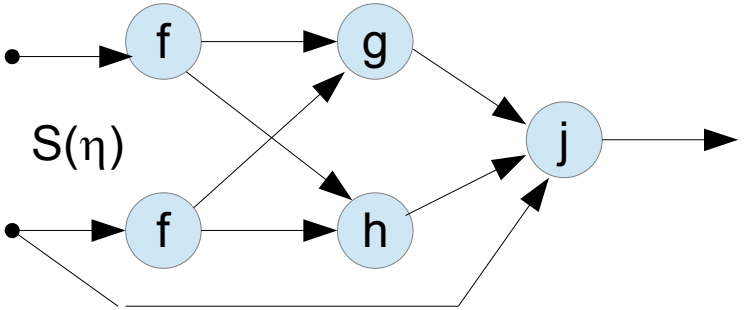


S: a composition scheme (dir.ac.graph w.IO)
C: set C of compute tokens in S

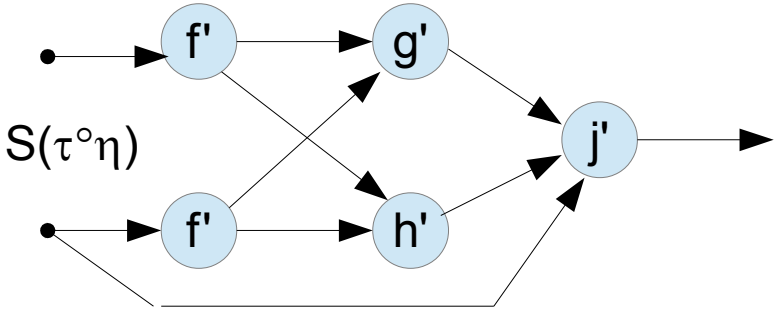
Every mapping
η: C → O , O a set of types of cp. operations
determines an algorithm S(η) for a function F_η.

Functorial substitution for a cp. assignment
τ:O→O'
transforms the algorithm S(η) to S($\tau\circ\eta$)

Generally, $F_\eta \notin O$ or S($\tau\circ\eta$) is not alg. for τF_η.



→



f'=τf etc.

.. algorithm computing F_η

.. algorithm computing $F_{\tau\circ\eta}$ $(\neq \tau F_\eta)$

0. Mapping operations and functions to multiple data

Map operations/functions f: $P \to Q$ to f': $P^n \to Q^n$, $(p_0,..,p_{n-1}) \to (f(p_0),..,f(p_{n-1}))$

.. for $P=A \times B$, f has two arguments, and f' is defined to be a function on $A^n \times B^n$.
or more $\approx (A \times B)^n$

Example: the scalar add operation +:$IR \times IR \to IR$ extends to the vector add operation.
... mpy ... * … to multiplying vectors by components

LISP: use MAPCAR for this extension

π–Nets: automatic extension (overloads operations/functions to more tuple operations)

.. more options: $f(a, b) = (f(a,b_0), \dots , f(a,b_{n-1}))$ for $a \in A$ and $b \in B^n$ etc.

This substitution is (an extension of) a true functor. Substituting all operations in an algorithm A without branches for a function f by the corresponding tuple operations yields an algorithm A' for the function f' on tuples.

Equivalently, the individual $f(p_i)$ can be computed separately which also allows for a control flow with individual branch selections, i.e. by horizontally expanding the original algorithm instead of substituting the operations/functions therein.

1. Automatic differentiation

$f: U \subset V \to W$ differentiable, with $V=\mathbb{R}^n$, $W=\mathbb{R}^m$ (real vector spaces)

$Df: U \to L(V,W)$, $Df(u)$ derivative of f in u (the approximating linear map)

Sum:	$D(f+g)(u)=Df(u) + Dg(u)$	for $g: V \to W$
Product:	$D(h(f,g))(u)=h(Df,g)(u)+h(f,Dg)(u)$	for $g: V \to Y$, $h: W \times Y \to Z$ bilinear
Chain rule:	$D(g \circ f)(u)=Dg(f(u)) \circ Df(u)$	for $g: W \to Z$
	$D(h(f+g))(u)=Dh(f(u),g(u)) \circ (Df(u)+Dg(u))$	for $g: V \to W$, general h

→ derivatives of compositions can be computed from the $(f(u),Df(u))$ of the components
chain rule involves composition of linear maps (matrix product)
→ can substitute calls to functions f by calls to the (f,Df) evaluated according to the rules
at individual $u \in U$ to compute composition of function tables with functions/operations

Applications only requiring particular partial derivatives $Df(u) \cdot v$ don't need matrix products:

Define $TU=U \times V$ (the tangential bundle over U, i.e. the union of the $T_u U=\{u\} \times V$) and
$Tf: TU \to TW$ $(u,v) \to (f(u),Df(u) \cdot v)$

The the composition rules simplify; the chain rule e.g. becomes $T(g \circ f) = Tg \circ Tf$.

→ can substitute calls to functions f by calls to the Tf evaluated with the simplified rules
(need to know the derivatives of the used elementary functions .. their approximations)

.. derive Hamiltonian vector fields from function

Automatic differentiation extends to higher derivatives:
Algorithms for functions $f: U \to W$ extend to functions on 'jet' spaces $U \times L(V,W) \times L_s(V^2,W)$,
or as $T(Tf)(u,v,x,y) = (f(u), Df(u)v, Df(u)x, Df(u)y+D^2f(u)(v,y))$.

πN: use the name T'f for an algorithm f or a function table and call to f or to T'f as needed.

Higher order approximations through higher order derivatives:

For $U \subset V$, $TU = U \times V$ is the set of first order approximations $\gamma(0)+t\cdot\gamma'(0)$ to curves $t \to \gamma(t)$ at 0.

For $f: U \to W$, Tf maps the approximation to γ to the approximation of $f\circ\gamma$ at 0. $T(f\circ g)=Tf\circ Tg$.

Define $T_{(r)}U = U \times V \times \ldots \times V$ ($r+1$ factors). Then $TU=T_{(1)}U$. $T_{(r)}U$ is the fiber sum r times TU.

The $(u,v_1,\ldots,v_r) \in T_{(r)}U$ parametrize the r^{th} order approximations $t \to u+t\cdot v_1+t^2/2\cdot v_2+\ldots+t^r/r!\cdot v_r$ to curves γ at 0, i.e. $v_i=\gamma^{(i)}(0)$.

Define $T_{(r)}f: T_{(r)}U \to T_{(r)}W$ to map the r^{th} order (Taylor) approximation to γ to that of $f\circ\gamma$ at 0.

Lemma: For fixed u, $(T_{(r)}f)_i$ is polynomial in the v_j and the derivatives of D^jf for $j \le i$.
$T_{(r)}$ is functorial, $T_{(r)}(f\circ g)=T_{(r)}f \circ T_{(r)}g$.

Proof: $(u,\gamma^{(1)}(0),\gamma^{(2)}(0),\ldots)=T_{(r)}\gamma(0,1,0,0,\ldots)$, need to compute $T_{(r)}(f\circ\gamma)(0,1,0,0,\ldots)$, the $(f\circ\gamma)^{(i)}(0)$.

$(f\circ\gamma)^{(1)}(0) = Df(u)v_1$ with $u=\gamma(0)$, $v_1=\gamma^{(1)}(0)$, $(f\circ\gamma)^{(2)}(0) = D^2f(u)(v_1,v_1) + Df(u)v_2$,

$(f\circ\gamma)^{(3)}(0) = D^3f(u)(v_1,v_1,v_1)+3D^2f(u)(v_1,v_2)+Df(u)v_3$ etc. by the chain rule … q.e.d.

Rem.: - for a diffeomorphism f $T_{(r)}f$ is a diffeomorphism, yet nonlinear on the fibers for r>1

- the functoriality of $T_{(r)}f$ allows for an automatic r^{th} order differentiation $\to D^rf(v_1,..,v_1)$
- can be applied to numerically 'solving' first order ODEs.

2. Substituting predefined operations by composite ones

Vector operations (e.g., linear mappings) and polynomial operations (evaluation, product, division) are defined in terms of the '+' and '*' operations of some base field/ring. Their substitution can be defined by a change of these ring operations.

From a given type of real numbers, complex operations can be defined on pairs of real numbers in the usual way, using the definition (a,b)*(c,d)=(ac-bd,ad+bc) as an algorithm. Complex n-vectors can be represented as size 2*n tuples, and the operation of multiplying a complex scalar to a complex n-vector becomes an operation multiplying a pair to a 2n-tuple with a 2n-tuple result. The real pair-to-vector multiplication can be overloaded with this complex operation without affecting the real scalar to 2n-vector operation.

This can be used to expand existing predefined linear and polynomial operations to other base rings while maintaining their usual names and calling syntax. The Substitution is thus applied to the defining algorithm of the predefined operation. The substitution of the operations within a previously defined program function is rarely needed, however, and does not need to be supported at all.

πN: Complex and quaternion operations are easily defined and packed into corresponding optypes; the same applies to other real algebras and even to redefining the real '+' and '*' by modulo operations. Algorithms selecting an optype overloading the '+' and '*' operations then dispose of the corresponding vector and polynomial operations as well.

Whether a 2n-tuple 'is' a real or a complex vector depends on the operations (optypes) applied to it only.

.. coord sel.

1) For 2n-tuples x,y their dot product 'x y' is the real number

$$\sum_i x_i \cdot y_i \ .$$

If 'cpx ·' is defined and 'cpx' selected then 'x y' denotes the complex dot product. x,y become accessed as tuples of pairs x:2,y:2 .

2) 2n-tuples are applied as polynomials to a number x writing 'x.pol y' to get

$$\sum_i x_i \cdot y^i \ .$$

After selecting 'cpx' 'x.pol' applies to pairs z, using pairs (x_{2i},x_{2i+1}) as complex coeffs.

3) n^k-tuples x are applied to k-tuples of numbers as polynomials in k variables writing 'x:k:n.pol y' to get

$$\sum_{(i,j,k..)} x_{(i,j,k,..)} \cdot y_0^{\ i} \cdot y_1^{\ j} \cdot y_2^{\ k} \ \ldots \ .$$

Selecting '+' and '·' operations mod(2) and n=2 2^k-tuples can be applied as Boolean functions to k-tuples with 0/1 entries.

.. bad BF encoding .. many other examples .. finite fields etc.

3. Using functorial substitution to support various number codes

Numbers need to be encoded by bit strings before they can be digitally computed with.

digit

enc: IR → B*	encoding function		partially defined
dec: B* → IR	decoding function		partially defined such that
rnd = dec°enc: IR → IR	the 'rounding' function	fulfils	enc(r) = enc(rnd(r))
		hence	rnd°rnd = rnd

Rem.: rnd ist unique, rnd/enc often derive from dec

Operations op: IR → IR are substituted by op'=enc°op°dec: B* → B* on the machine

r = rnd(r) ⇒ dec(op'(enc(r))) = rnd(op(r)) (substitution inserts rounding)

Operations op: IR×IR → IR etc. are handled similarly.

Numeric algorithms (compositions of real operations) are executed on the machine by performing functorial substitution of every operation op on the reals by the corresponding op' on number codes rounding its output. The same substitution also applies to the composed real function. In general, substituting within the algorithm does not yield an algorithm for the substitute of the composed function without intermediate rounding (the 'fused' one).

If a program function g is called within an algorithm for f, g is not substituted as a fused composite operation with added rounding, but by the function computed by the substituted algorithm for g similarly to expanding the algorithm for g into the algorithm for f, flattening the hierarchic definition. Machines may be built to compute certain fused functions, however.

Standard PLs distinguish different types of numbers by the codes to be applied (**int**,**float**).
.. no abstract reals, rounded substitutes for the real operations only
.. mathematical algorithms substituted by hand (conscientiously ..?)

π-Nets: supports several standard and non-standard encodings including

I32, X16, X35, V144, F32, F64, G45 .. to be expanded

by their (unique) rounding operations only and as attributes to the abstract computations performed by its processes telling the compiler how to substitute operations. The rounding operations are available as predefined real operations allowing to test for the applicability of the code and to model the operation of the executing machine. For every encoding, errors can and need to be estimated and tracked through an algorithm, maybe using simulation.

The I32 code applies to integers only and does not perform rounding at all. An extra integer divide operation is available to explicitly 'round' real numbers to nearby integers. The functorial substitution by rounded operations also applies to the comparisons like 'a=b'. It becomes the comparison of the codes of a and b. Invalid data are encoded as well.

By default, number codes extend to tupels of real numbers using tuples of codes of their entries. n-Tuples can be applied as functions to integer indexes in the range 0...n-1 only (yield invalid data otherwise). As primary data tuples can use codes of their own, e.g. an n-tuple of fixed point mantissas plus a extra exponent. The elementary functions typically add extra approximation errors. Certain composite operations can be implemented as 'fused' operations w/o intermediate roundings, e.g. the complex operations for V144.

Rounding error estimate .. constant/var .. ähnl. optype const .. err fct

Implementation in the π-Nets compiler:

- intermediate code (CDGF) DFTs represent real number operations
  may use associativity/commutativity for optimization etc.

  tuple operations not expanded into scalar operations, appear as tuple DFTs

  no expansion of function calls in the intermediate code

  number code selection is a CFDG attribute

- automatic insertion of rounding operations for interpretation of the intermediate code for the purpose of simulation

  interpretation applied to constant folding and to external processes uses a high precision code type performing rounded operations from which the less precise application codes are rounded ($rnd_{app} = rnd_{app} \circ rnd_{max}$)

  automatic substitution of encoded operations during code generation and

  automatic insertion of code conversions for data transfers between processes

- storage for intermediate data and the variables of application processes:
  storage of numbers is in full internal precision for interpretation/simulation, but is for the required code word size only on the processors executing native code

  Rem.: external memory is a set of special function nodes storing/xferring msgs

Error handling and 'invalid' codes.

The automatic insertion of rounding operations during simulation or code generation may cause runtime errors at places that are not explicit in the program, such that the errors are not handled by the program. The same kind of error also occurs for the implicit code conversions when data are sent from some process to another one using different codes. In these cases the failing rounding operation (indicating that the number to be encoded is outside the encoded domain) returns an 'invalid' but does not break execution. Embedded applications often tolerate erroneous data.

All scalar operations on numbers are defined to yield an invalid result on invalid input. In a tuple, individual entries may be valid or invalid, and tuple operations defined in terms of scalar operations can still deliver tuples with some valid components. In order to maintain this functionality, number codes must provide at least one bit pattern for invalid data. The standard floating point formats provide NaN codes that can be used for this, and standard floating point units deliver NaN outputs on NaN inputs. If more codes are available for invalid data, they can be used to transport more information on the error.

As π–Nets only provides numeric operations and numeric or invalid output there seems to be no way to symbolically annotate its numeric outputs on some display. A simple trick is played to overcome this limitation. Strings like “abc” are allowed as literals for invalid data and sent to the predefined process writing to the display. This process interprets the invalid data as the command to print out the string or the error it represents.

4. Deriving error functions

Rounding errors reflect the approximate representation of input data and the results of operations. They are usually estimated by an upper bound but actually depend on the arguments of the rounded operation. For a program function, errors accumulate depending on the data (also via the branches taken that may depend on rounding). During simulation operations may be substituted by double operations one executed with the prescribed rounding and the other at the highest available precision.

A similar approach can be taken to analyze the approximation errors by simulation. They occur when the true result is the limit of a sequence of numbers computed by a recursive program function stopped after a finite number of steps. Ideally, the recursion delivers nested intervals around the limit. Achieving a desired precision will also depend on the precision of limits on the way and on the rounding errors after substitution.

If a finite computation aims at approximating a function f on an infinite domain, e.g. an open subset U of some real vector space V, the result can only be a finite tuple of input parameters to some fixed algorithm computing the approximate values. A standard setup is to use a discrete subset $H \subset U$, to compute approximate values of f on H, and to apply an interpolation operation on these to get the approximation to f on U. More generally, there is a finite-dimensional parameter space W, a linear 'interpolation' operation

$$d: W \to F(U) \qquad \text{and a (generalized) sampling operation} \qquad c: F(U) \to W$$

such that $c \circ d = id_W$. This is similar to the coding of numbers. The 'rounding' $e = d \circ c$ can serve to substitute operations p of F(U) (like differentiation) by operations $p' = c \circ p \circ d$ on W.

Examples:

0) W is the space of functions on H, F(U) → W is restriction. The interpolated functions for a subspace P⊂F(U) are in bijection to W under restriction.

1) U is the unit cycle in the complex plane and H is the set of n-th roots of unity. Interpolate with the (sin x)/x kernel. P is the space of complex polynomials of degrees < n and in bijection with the space of their coefficients (DFT).

2) U is the sphere in $\mathbb{R}^3$, W the space of homogeneous polynomial functions of degree n on $\mathbb{R}^3$. Restriction to U is injective.

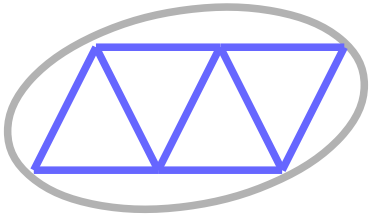
3) $F_r = F(U, \Lambda^r V^*)$ r-forms on U, $d: F_r \to F_{r+1}$ exterior differentiation

$\int_M : F_r \to \mathbb{R}, \theta \to \int_M \theta$ integration over r-dim. M⊂U

Stoke's formula: $\int_M d\theta = \int_{\partial M} \theta$ r+1-dim. M with boundary ∂M

Discretization: describe M⊂U as a simplicial complex, $M = \bigcup_i S_i$

'encode' θ by the tuple of the $\int_N \theta$ over all r-subsimplexes of the S_i



→ can compute integrals of θ over arbitrary r-submanifolds of M exactly from its 'code'

→ get encoded d operator by using Stokes formula to get the code for d (all $\int_{Si} d\theta$)

can use Whitney forms for interpolation

Remarks on the d-operator on forms and its discretization:

- The d-operator on forms does not depend on a choice of coordinates; the same holds for differential derived from d.
  The discretization transforms such operators to linear ones on tuples

- a prominent operator of this kind is the Laplacian on k-forms,

  $$\Delta\theta = *d*d\,\theta + d*d*\,\theta$$

  that also involves the Hodge-Operator $*: \Lambda^r V^* \to \Lambda^{n-r} V^*$ (n=dim V). '*' needs to be 'encoded', too. The corresponding 'code' tuple spaces W_r and W_{n-r} are not isomorphic, however, as they should be. Instead, a dual cell complex and a corresponding d operator need to be used. The n-r-cells in the dual complex correspond 1-1 to the r+1 simplexes to which they are orthogonal w.r.t. the Euklidean dot product.

- The Maxwell equations describing electrodynamics can be expressed with the d operator. E and H are the electric and magnetic fields on IR^3, respectively, D and B the dielectric and magnetic flows. E and H are time dependent 1-forms on IR^3, D and B time dependent 2-forms, and D=*E, B=*H up to constants. Computations of these forms can then use the discrete exterior calculus (DEC). This application and others legitimate its support by special operations.

  (see Bossavit, Leok/Stern, Arnold)

Completing π-Nets

- syntax for literals, expressions, control
  definitions of constants, functions
  'optype' definitions with options to overload, inherit

- 'atype' definitions (types of automata with memory), options to overload, inherit
  process definitions (w. variables, sub automata); automata are primitive processes
  communications between processes incl. external I/O
  real/execution time control, sub processes (threads), rounding/code selection

---

- 'ntype' definitions of target components w. inheritance (hierarchy), P,B,M,A
  .. FPGA components are defined as special composite components

- 'node' definitions describing the target network (incl. data links and boot trees)

.. structural 'program' for target network .. similar automata and processor networks

- syntax to assign threads to processors, some caching&reconfiguration support
  .. process variables located on the executing P or M nodes

An example: Musical Processes

```
const     t1 288, t4 264, t8 260, t38 268, t16 258, t316 262,
          c 60, e 64, g 67, c' 72,   band 151, pia 128, wood 243,   x0 384

fct 1 1 fib                .. defines algorithm for Fibonacci sequence mod(24)
{     → i
      { → aa,bb,ii                                   ,, end recursive Λ expression
            if    ii=0   then     aa
      else  if    ii=1   then     bb
      else                        bb, aa+bb %24, ii-1  ←} (0,1,i)
}

apc mt on host { $$+0.125   1 >> R  ←}        .. increase musical time

apc mus  on host,                          .. main application process group
           (mdo) vc1    band, 55           .. midi sub automata (predef.type)
           (mdo) vc2      pia, 77, x0                .. receiving initial input
           (mdo) vc3    wood, 88           .. midi threads redefine top lvl.'apply'
{
  #vc1    t1      24:{→i   fib(2i)+66 t16      fib(2i+1)+66 t316 }
  #vc2    14:{→i  fib(3i)+54  fib(3i+1)+58 t38   fib(3i+2)+60 t8}   c,e,g,c' t2
  #vc3     7:{   c' t316   c' t316   c' t4   c' t316   c' t316  }    drum 100  c' t38
}
```

2 charged particles moving on the unit circle

```
const dt 1/256

fct 2 2 chg {→ gh,k   if gh>3  then  gh-π,1-k    else if gh< -3 then  gh+π,1-k   else  gh, k }
fct 2 2 pos {→ s,k     if k=0    then  (cos s, sin s)   else (- cos s, - sin s) }
fct 2 1 ang {→ g,h  {→d  if d>π then d-2π ←  else if d< -π then d+2π ←  else d }(g-h) → e   e/2 }

apc (gc) gd on host                                                    .. display

apc part on host,  t 0,  g 0,  g' 0.5,  gk 0,  h 0,  h' 1.5,  hk 1
{
      $$+ dt                                                           .. real time ! .. unit is 'sec'

      pos(g,gk) → gx,gy              pos(h,hk) → hx,hy                 .. in IR²
      ang((g+π*gk),(h+π*hk)) → a
      cos(a)/((hx-gx)²+(hy-gy)²) → f                                   .. repelling force
      { if a<0 then -f else f } → gh''

      g + g' dt + gh'' dt²/2   >> g                                    .. differential equation
      g' + gh'' dt             >> g'
      h + h' dt – gh'' dt²/2   >> h
      h' – gh'' dt             >> h'                                    .. same for both coordinates

      chg(g,gk) → ng,ngk     ng >> g    ngk >> gk
      chg(h,hk) → nh,nhk     nh >> h    nhk >> hk                     .. change of coordinates

      0,0,0 >> gd.r                                                    .. display g,h

      100(gx,gy)+(110,110) >> gd.p      0,0,0 >> gd.b
      100(hx,hy)+(110,110) >> gd.p      0,0,0 >> gd.b
←}
```