Open Source SPIRAL 8.1
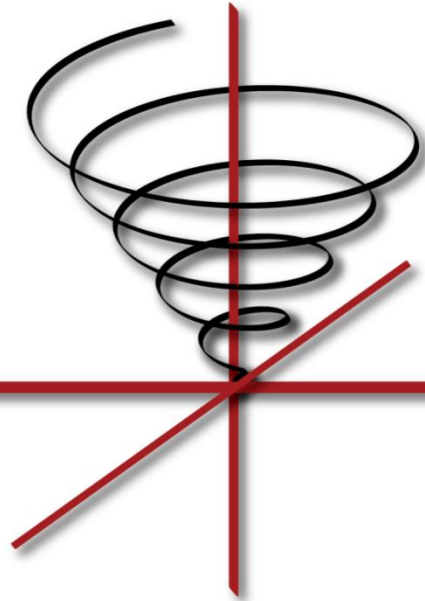# System Walk-Through

Tutorial at HPEC 2019
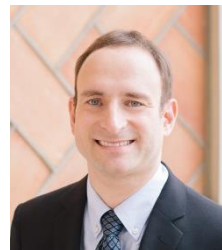
Franz Franchetti
Tze Meng Low
Carnegie Mellon University

Mike Franusich
SpiralGen, Inc.

Franz Franchetti   **Tze Meng Low**   **Mike Franusich**

Tutorial based on joint work with the Spiral team at CMU, UIUC, and Drexel

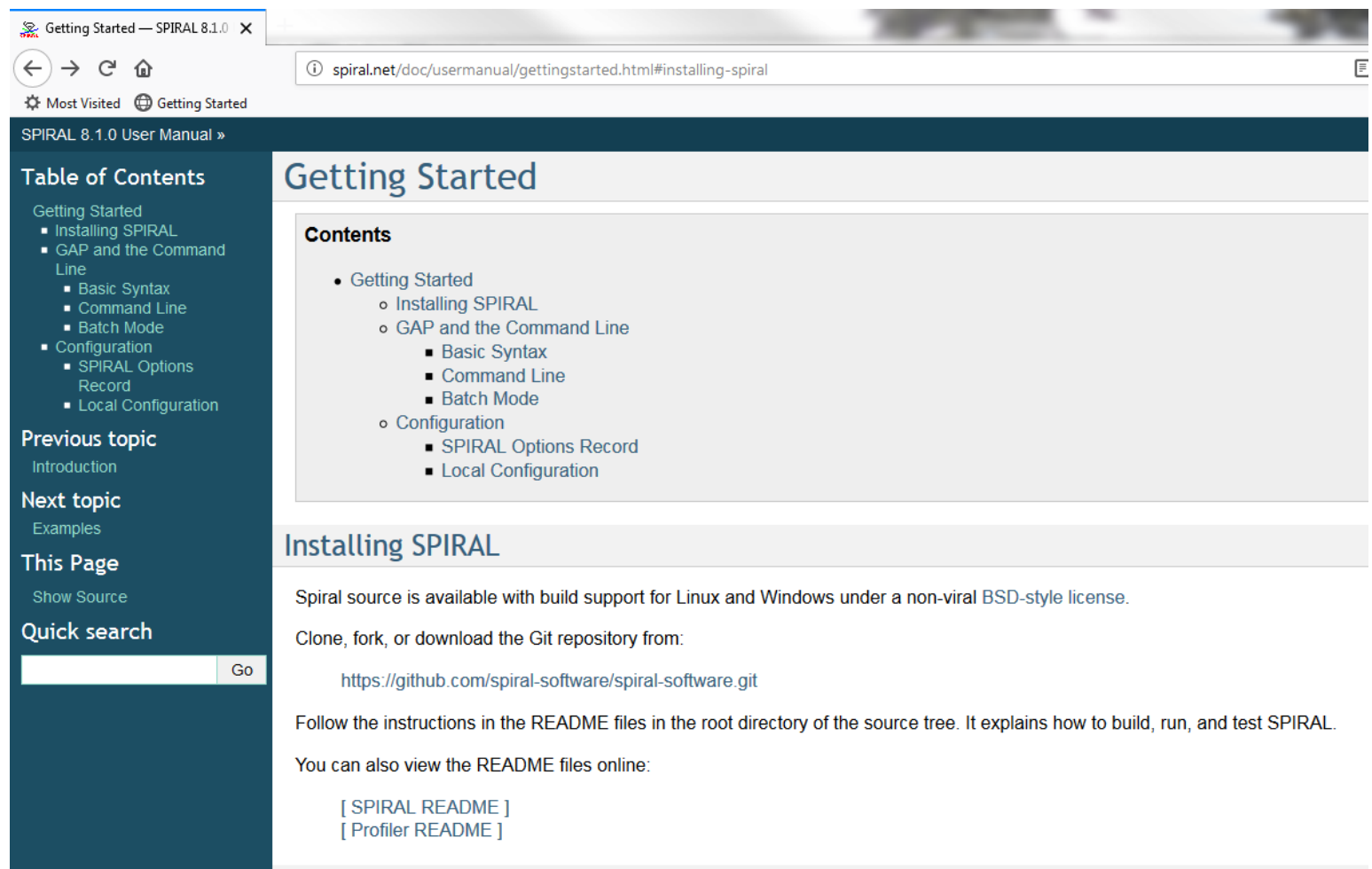# Organization

- **Installation**

- GAP3 language

- Spiral objects and data types

- Spiral infrastructure

- Special hardware: SMP/OpenMP

- Special hardware: SIMD/AVX

- Benchmark/production code infrastructure

Installing the Open Source Spiral Version

http://www.spiral.net/doc/usermanual/gettingstarted.html#installing-spiral

System Startup

Starting and exiting

```
C:\Spiral\spiral-core> spiral.bat
spiral> quit;
```

Break:	`^C`
Exit:	`^D`
Command completion:	`<TAB>`
Record fields:	`^W`

Command line options: `spiral.bat`

```
@echo off
set SPIRAL_DIR=%~dp0
set GAP_MEM=2048m

Rem uncomment for debugging profiler
Rem set PROFILER_LOCAL_ARGS=--debug --keeptemp

set SPIRAL_CONFIG_SPIRAL_DIR=%SPIRAL_DIR%
set SPIRAL_CONFIG_PATH_SEP=\

"%SPIRAL_DIR%gap\win32\Release.64\spiral.exe" -m %GAP_MEM% -x 1000
-l "%SPIRAL_DIR%gap\lib" "%SPIRAL_DIR%_spiral_win.g"
```

Examples (1)

FFT: Scalar C code

```
opts := SpiralDefaults;
transform := DFT(4);
ruletree := RandomRuleTree(transform, opts);
icode := CodeRuleTree(ruletree, opts);
PrintCode("DFT4", icode, opts);
```

FFT: AVX 4-way double-precision C + intrinsics

```
opts := SIMDGlobals.getOpts(AVX_4x64f);
transform := TRC(DFT(16)).withTags(opts.tags);
ruletree := RandomRuleTree(transform, opts);
icode := CodeRuleTree(ruletree, opts);
PrintTo("AVX_DFT16.c", PrintCode("AVX_DFT16", icode, opts));
```

Examples (2)

FFT: 2-way OpenMP Multi-Threaded SSE2 Code

```
opts := LocalConfig.getOpts(
            rec(dataType := T_Real(64), globalUnrolling := 512),
            rec(numproc := 2, api := "OpenMP"),
            rec(svct := true, splitL := false, oddSizes := false,
                stdTTensor := true, tsplPFA := false));
transform := TRC(DFT(32)).withTags(opts.tags);
ruletree := RandomRuleTree(transform, opts);
icode := CodeRuleTree(ruletree, opts);
PrintTo("SSE_OMP2_DFT32.c",
        PrintCode("SSE_OMP2_DFT32", icode, opts));
```

FFT: 4-way AVX C + intrinsics, Dynamic Programming

```
opts := SIMDGlobals.getOpts(AVX_4x64f);
transform := TRC(DFT(512)).withTags(opts.tags);
best := DP(transform, rec(), opts);
ruletree := best[1].ruletree;
icode := CodeRuleTree(ruletree, opts);
PrintTo("AVX_DFT512.c", PrintCode("AVX_DFT512", icode, opts));
```
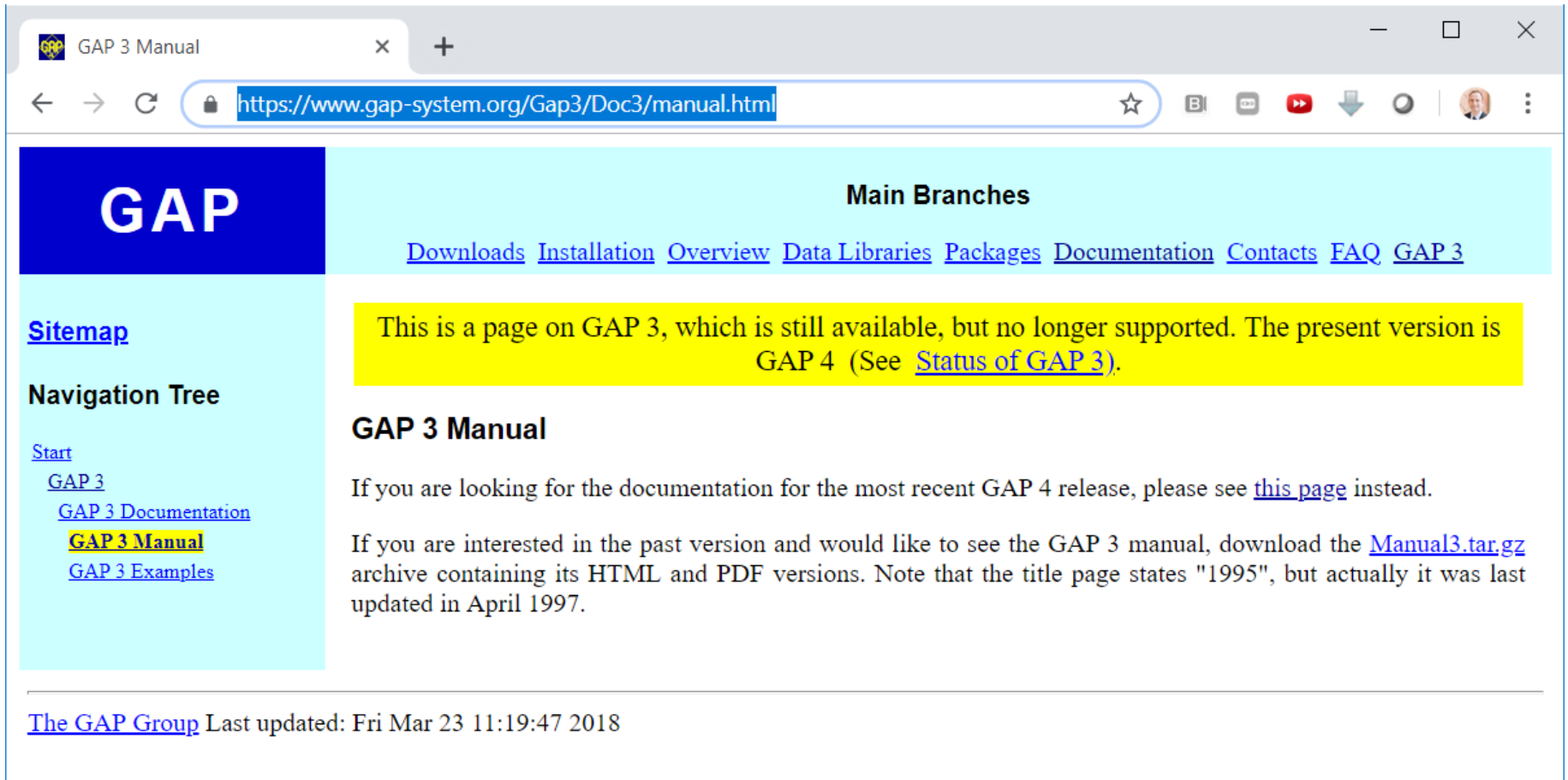
Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

GAP3 Manual

https://www.gap-system.org/Gap3/Doc3/manual.html

GAP3 and GAP/Spiral Data Types

GAP scalars

```
1; -1;                      # integer
1.1; -1.543;                # IEEE floating point (GAP/Spiral specific)
3/4;                        # rational number
Double(3/4);                # typecast to floating-point
true; false;                # logical values
Cplx(1, 1);                 # complex number: 1 + i
E(4);                       # Cyclotomic numbers: complex root of unity
```

Basic arithmetic/logical operations

```
1 + 2; 1 + 2.0; 1 + 3/4;
1.1 * Cplx(1, 1);
true and false;
Log2Int(8192);
Re(E(4)); Im(E(4));    # Real and imaginary parts
True; False;           # function returning true/false
```

Transcendental functions

```
SinPi(1.5);                 # sin(1.5 * M_PI) -> floating-point number
CosPi(3/4);                 # cos(3/4*\pi) -> cyclotomic number
Sqrt(2);
Sqrt(2.0);
```

GAP3 Assignment and Functions

Assignments

```
x := 1;                    # assign a <- 1
x = 1;                     # check equality
x < 2; x > 0; x <> 7;      # inequalities

Unbind(x);                 # delete symbol
let(a := 5, a+3);          # local symbol
last;                      # last result
last2; last3;              # and even farther back
```

GAP functions

```
a -> 1;                    # maps a to 1
a -> a + 1;                # maps a to a+1
() -> 1;                   # no argument, returns 1
(a, b) -> a + b;           # adds the two arguments a and b
f := (n) -> When(n = 1, 1, n * f(n - 1));     # recursive function
ApplyFunc(f, [3]);      # apply function to parameter list
```

GAP3 Control Flow

Procedures, procedural functions

```
add1 := function(n)
   local m;
   m := n + 1;
   return m;
end;


vaarg:= function(arg) return Length(arg); end;
vaarg(1, 2); vaarg(1, 2, 3);  # variable number of arguments
```

Loops

```
for i in [1..5] do Print(i); od;
i:= 5; while i > 0 do Print(i); i := i-1; od;
DoForAll([1..5], PrintLine);
```

Conditionals

```
a := 3;
b := When(a < 3, a+1, 2*a);
if a < 3 then c := a+1; else c := 2*a; fi;
c;
a -> Cond(a<0, 0, a>10, 20, 2*a);     # functional switch statement
```

GAP3 Lists

Lists

```
[];                                # empty list
[1..4];                            # dense list
[1, 3, 4000];                      # sparse list
[[1, 2], [3, 4]];                  # matrix = list of lists
[[[1, 2], [3, 4]], [[5, 6], [7, 8]]];      # rank 3 tensor
[1, rec(), Set([]), ()->3];    # mixed data type lists
```

List operations

```
[1, 2]::[4..7];                    # concatenation
Concat([1, 2], [4..7]);            # concatenation
List([1, 2, 4, 8], Log2Int);    # apply function to list elements
Map([1, 2, 4, 8], Log2Int);     # apply function to list elements
Length([1..5]);                    # length of list
1 + [2..5];                        # pointwise arithmetic
FoldL([1..3], (a,b)->a+b, 0);  # Fold left operation
FoldR([1..3], (a,b)->a-b, 0);  # Fold right operation
Cartesian([1..4], [3, 4]);      # Cartesian product
ForAll([1..4], IsInt);          # Apply function, then reduce with AND
ForAny([1..4], IsInt);          # Apply function, then reduce with OR
Filtered([1..50], IsPrime);     # Apply filter function, drop if false
1 in [1..5];                       # membership test
```

GAP3 Sets, Vectors and Matrices

Sets

```
Set([]);                          # empty set
Set([1, 2, 2, 3, 4]);             # Elements in sets are unique
```

Set operations

```
s := Set([]);
AddSet(s, 3);
AddSet(s, 3);
AddSet(s, 4);
s;
SubtractSet(s, [3]);
s;
```

Vectors and matrices

```
v := [1..3];                      # list is a vector
m := [[ 1..3], [2..4], [3..5]];   # matrix
m * v;                            # matrix * vector
v * m;                            # vector * matrix
TransposedMat(m);                 # transpose matrix
m * [[1],[2],[3]];                # column vector
[[1..3]] * m;                     # row vector
```

GAP3 Strings

Strings: as in C

```
"";                              # empty string
'a';                             # char
"abc";                           # string
"a\tb\nc";                       # control characters
```

String operations

```
"abc"::"abc";                    # concatenation
Concat("abc", "abc");            # concatenation
StringList("abc");               # string -> list of char
StringToUpper("abc");            # -> upper case
StringToLower("aBc");            # -> lower case
StringInt(5);                    # integer -> string
```

GAP3 Records

Records

```
rec();                            # empty record
rec(key1 := 2, key2 := 17);       # key/value pairs
r := rec(a := 1, b := "2",        # record of records
        c := rec(a := 1), d:= []);
r.a;                              # direct access
i := "a";                         # record field name
r.(i);                            # indirect access
```

Record operations

```
RecFields(r);                     # List of record fields
IsSystemRecField("__doc__");      # Some are system fields
CopyFields(rec(a := 1, b := 2),   # merge records
   rec(b := 3, c := 4));
```

GAP/Spiral Classes

Class definition: uses SELF (class and instance are both records)

```
Class(A, rec(
    __call__ := meth(self, v) return WithBases(self, # copy base fields
        rec(a := v,                 # initialize values
        operations := rec(          # need that to print with state
            Print := self >> Print("A(", self.a, ")")))); end,
    a := 1,                         # state of the object
    geta := self >> self.a,     # access functions: get
    seta := meth(self, v) self.a := v; end, # access functions: set
));
```

Class operations

```
a := A(3);              # instantiate A by calling constructor
a.seta(a.geta()+1);    # use access functions
a.a;                    # updated value of a.a
a.name;                 # Class name
a.__bases__;            # base classes
a.__bases__[1].a;       # get value of a from the base class
Unbind(a.a);
a.a;                    # refers to the base class value
```

File I/O

```
Print("Hello World!");
LogTo("log.txt");                   # log to log.txt
PrintLine("Numbers: ", 1, ", ", 2);
PrintTo("file.txt", "a := ", 2);
AppendTo("file.txt", ";\n");
Read("file.txt");                   # read file into stdin
Exec("del file.txt");               # MS/DOS system call
a;                                  # file contents was executed
LogTo();                            # Turn off logging
```

GAP/Spiral Packages and Name Spaces

Load tree: `init.g`

```
RequirePackage("arep");
Package(spiral);
Include(config);
Include(trace);

...
Load(spiral.rewrite);
Load(spiral.code);
ProtectNamespace(code);
Declare(CMeasure);

...
```

Packages and name spaces commands

```
avx.addsub_4x64f;
Import(avx);
addsub_4x64f;
avx.addsub_4x64f := false;
addsub_4x64f;

Dir(avx);
Info(addsub_4x64f);
Doc(DP);
```

GAP/Spiral Debugging

Stack-based debugger

```
Error("msg");
f := (n) -> When(n = 1, Error("at bottom"), n * f(n - 1));
f(10);
Top();
n;
Down();
n;
Down();
n;
Up();
n;
n + 1;
```

Spiral Configuration

Spiral options record

```
opts := SpiralDefaults;
opts.globalUnrolling;
opts.arrayBufModifier;
opts.includes;
opts.unparser;
opts.codegen;
opts.useDeref;
opts.Xtype;
opts.compileStrategy;
opts.breakdownRules;
...
```

Local configurations: set via _spiral.g

```
LocalConfig;
LocalConfig.getOpts(SSE_4x32f);
LocalConfig.cpuinfo;
LocalConfig.gapinfo;
LocalConfig.osinfo;
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**
  Types, values and expressions

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Spiral Values and Data Types

Value wrapper and object

```
V(1);
v := V(1.1);
_unwrap(v);
v.v;
v.t;
```

Scalar data types

```
Value(TReal, 1.0);
Value(TDouble, 1.0);
Value(T_Real(32), 1.0);
Value(T_Real(64), 1.0);
Value(TInt, 1);
Value(T_Int(32), 1);
Value(TComplex, Cplx(0, 1));
```

Spiral Array Data Types

One and multi-dimensional Arrays

```
t := TArray(TReal, 4);                    # double[4]
v := Value(t, [1,2,3,4]);                 # double[4] = {1.0,2.0,3.0,4.0}
mt := TArray(TArray(TReal, 4), 4);
Value(mt, Replicate(4, v));               # initialize a 4x4 matrix
```

Pointers

```
TPtr(TReal);                              # *double
```

Spiral Variables and Expressions

Variables

```
i := Ind();                              # integer index
j := Ind(4);                             # index, 0 <= j < 4
k := var.fresh_t("j", TInt);             # create a "fresh" variable
var.table.(v.id);                        # global variable table
```

Expressions

```
a := var.fresh_t("a", TReal);            # a few real variables
b := var.fresh_t("b", TReal);
c := var.fresh_t("c", TReal);
a + b;                                   # + is overloaded
add(a, V(2.0));
a + 2;
e := add(a, b);                          # use add function
e.args;                                  # the operands
e.t;                                     # expressions carry a type
g := add(a, b, c);                       # not just binary
h := add(a, mul(b, c));                  # expressions
k := add(neg(a), mul(b, c, V(1.1)));     # expressions
mul(V(1.0), V(2.0));                     # evaluates at construction
sub(V(1), V(2.0)).t;                     # type unification
```

Spiral Comparisons, Conditionals, Etc.

Comparisons

```
i := Ind();                          # an integer variable
j := Ind();                          # an integer variable
eq(i, V(1));                         # i == 1
leq(i, j);                           # comparison: <=
geq(i, j);                           # comparison: >=
lt(i, j);                            # comparison: <
gt(i, j);                            # comparison: >
cond(leq(i, 0), 0, i);              # C ? : conditional assignment
min(i, j);                           # minimum
max(i, j);                           # maximum
```

Error handling

```
i := Ind();                          # an integer variable
errExp(TInt);                        # illegal value of type int
noneExp(TInt);                       # undefined value, type double
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**
  icode

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Spiral Abstract Code (icode)

```
program(
    chain(
        func(TVoid, "init", [  ],
            chain()
        ),
        func(TVoid, "transform", [ Y, X ],
            decl([ t57, t58, t59, t60, t61, t62, t63, t64 ],
                chain(
                    assign(t57, add(deref(X), deref(add(X, V(4))))),
                    assign(t58, add(deref(add(X, V(1))), deref(add(X, V(5))))),
                    assign(t59, sub(deref(X), deref(add(X, V(4))))),
                    assign(t60, sub(deref(add(X, V(1))), deref(add(X, V(5))))),
                    assign(t61, add(deref(add(X, V(2))), deref(add(X, V(6))))),
                    assign(t62, add(deref(add(X, V(3))), deref(add(X, V(7))))),
                    assign(t63, sub(deref(add(X, V(2))), deref(add(X, V(6))))),
                    assign(t64, sub(deref(add(X, V(3))), deref(add(X, V(7))))),
                    assign(deref(Y), add(t57, t61)),
                    assign(deref(add(Y, V(1))), add(t58, t62)),
                    assign(deref(add(Y, V(4))), sub(t57, t61)),
                    assign(deref(add(Y, V(5))), sub(t58, t62)),
                    assign(deref(add(Y, V(2))), sub(t59, t64)),
                    assign(deref(add(Y, V(3))), add(t60, t63)),
                    assign(deref(add(Y, V(6))), add(t59, t64)),
                    assign(deref(add(Y, V(7))), sub(t60, t63))
                )
            )
        )
    )
)
```

Spiral icode

Basics

```
c1 := skip();                        # NOP
a := var.fresh_t("j", TInt);         # create a "fresh" variable
c2 := assign(a, V(0));               # assignment
c3 := assign(a, fcall("foo"));       # call to foo()
c4 := chain(c1, c2, c3);             # basic block
i := Ind(4);                         # loop index
c5 := loop(i, 4, c4);                # loop
c6 := decl([a], c5);                 # declare a variable

PrintCode("", c6, SpiralDefaults);   # pretty print as C code
```

Internal fields

```
a.id; a.t;
c2.exp; c2.loc;
c3.cmds;
c4.var; c4.range; c4.cmd;
c5.vars; c5.cmd;
```

Spiral icode

Arrays and pointers

```
X;                                          # default input
Y;                                          # default output
X.t; Y.t;                                   # they are pointers
deref(X + V(4));                            # *(X+4)
t := var.fresh_t("t", TArray(TReal, 4));    # double[4]
nth(t, V(2));                               # T[2]
tcast(TInt, deref(X + V(4)));               # typecast
```

Constants

```
d := var.fresh_t("D", TReal);        # data variable
v := V(1.1);                         # value
c := data(d, v,                      # declare constant
        assign(nth(X, 0), nth(X, 0) * d)
    );
```

Spiral icode Definition

Expressions and commands

```
# spiral-core\namespaces\spiral\code\ir.gi
Class(neg, AutoFoldExp, rec(
    ev := self >> -self.args[1].ev(),
    computeType := self >> let(t := self.args[1].t,
        Cond(IsPtrT(t),
             t.aligned([t.alignment[1],
              -t.alignment[2] mod t.alignment[1]]), t)),
));


Class(chain, multiwrap, rec(
   isChain := true,
   flatten := self >> let(cls := self.__bases__[1],
       CopyFields(self, rec(cmds := ConcatList(self.cmds,
           c -> Cond(IsChain(c) and not IsBound(c.doNotFlatten),
                      c.cmds, ObjId(c) = skip, [], [c]))))),
   __call__ := meth(arg)
       local self, cmds;
       [self, cmds] := [arg[1], Flat(Drop(arg, 1))];
       return WithBases(self, rec(
           operations := CmdOps,
           cmds        := Checked(ForAll(cmds, IsCommand), cmds)));
   end
));
```

Looking Up The Language

Script to find all expressions, types, commands, etc.

```
# all objects defined in spiral.code
allobs := Dir(spiral.code);

# filter function, checking that base classes are in a given list
flt := bl -> (o -> (ForAny(bl, b -> b in let(ob := spiral.code.(o),
    When(IsRec(ob) and IsBound(ob.__bases__), ob.__bases__, [])))));

# list all expressions
Filtered(allobs, flt([Exp, AutoFoldExp]));

# list all types
Filtered(allobs, flt([AtomicTyp]));

# list all commands
Filtered(allobs, flt([Command]));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**
  Symbolic functions

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Lambda Functions

Definition

```
i := Ind(4);                          # variable with range
f := Lambda(i, i+1);                  # i -> i+1
j := Ind(4);                          # variable with range
g := Lambda([i, j], imod(i*j, 4));    # function in 2 variables
```

Operations on functions

```
f.at(0);                              # evaluate function
f.tolist();                           # create table for function
k := Ind(4);
Lambda(k, g.at(k, V(1)));             # partially evaluate g(.,1)
m := Ind(4);
h := Lambda(m, 2*m);                  # i -> 2*i
LambdaCompose(f, h);                  # function composition
```

Function properties/fields

```
f.domain();
f.range();
f.t;
f.vars;
f.expr;
```

Σ-SPL Index Mapping Functions

Definition

```
f := fId(4);                # I4->I4: i->i
j := Ind(4);                # variable with range
g := fBase(j);              # I1->I4: i->j
h := fAdd(4,2,1);           # I2->I4: i->i+1
u := L(16, 4);              # permutation i -> \ell(16,4)(i)
```

Operations on functions

```
f.at(0);                    # evaluate function
f.tolist();                 # create table for function
f.lambda();                 # convert to Lambda function
r := fTensor(f, g);         # tensor product of functions
s := fCompose(r, u);        # function composition
```

Function properties/fields

```
f.domain();
f.range();
```

Σ-SPL Diagonal Functions

Definition

```
f := fConst(4, 1.1);              # I4->R: i->1.1
g := dOmega(8, 2);                # f_N,k : N -> C : i -> omega(N, k*i)
h := FList(TReal, [1.1, 1.2, 1.4, 1.4]);     # table lookup
u := FData([V(1.1), V(1.2), V(1.3), V(1.4)]); # table lookup w/var
```

Operations on functions

```
g.at(3);                         # evaluate function
g.at(3).ev();                    # simplify the result
f.tolist();                      # create table for function
g.lambda();                      # convert to Lambda function
r := diagTensor(f, u);           # tensor product of functions
r.tolist();                      # what does diagTensor do?
s := fCompose(r, L(16,4));       # composition of permutation and
s.tolist();                      #    tensor product of functions
```

Function properties/fields

```
f.domain();
f.range();
u.var;
u.var.t;
u.var.value;
```

Defining Σ-SPL Symbolic Functions

fId and fBase

```
# spiral-core\namespaces\spiral\spl\perms2.gi
Class(fId, PermClass, rec(
    domain := self >> self.params[1],
    range  := self >> self.params[1],
    def    := size -> Checked(IsPosInt0Sym(size), rec(size := size)),
    lambda := self >> let(i := Ind(self.params[1]), Lambda(i,i)),
    transpose := self >> self,
    isIdentity := True
));


Class(fBase, FuncClass, rec(
    abbrevs := [ var -> Checked(IsVar(var) or
                ObjId(var)=ind, [var.range, var]) ],
    def    := (N, pos) -> rec(),
    domain := self >> 1,
    range  := self >> self.params[1],
    print  := (self, i, is) >> Print(self.name, "(",
        When(ObjId(self.params[2]) in [var, ind],
            Print(self.params[2]), PrintCS(self.params)), ")"),
    lambda := self >> let(i := Ind(1), Lambda(i, self.params[2]))
));
```

Defining Σ-SPL Symbolic Functions

fCompose

```
# spiral-core\namespaces\spiral\spl\perms2.gi
Class(fCompose, FuncClassOper, rec(
    domain := self >> Last(self._children).domain(),
    range := self >> self._children[1].range(),

    lambda := self >>
        FoldL1(List(self.children(), z->z.lambda()),
            _rankedLambdaCompose),

    transpose := self >> self.__bases__[1](
        List(Reversed(self.children()), c->c.transpose())),

    isIdentity := self >> ForAll(self._children, IsIdentity),
));
```

Defining Σ-SPL Symbolic Functions

dOmega and dLin for Twiddle Diagonal

```
Class(dLin, DiagFunc, rec(
    checkParams := (self, params) >> Checked(Length(params)=4,
        IsPosInt0Sym(params[1]), IsType(params[4]), params),
    lambda := self >> let(i:=Ind(self.params[1]),
        a := self.params[2], b := self.params[3],
        Lambda(i, a*i+b).setRange(self.params[4])),
    range := self >> self.params[4],
    domain := self >> self.params[1]
));

Class(dOmega, DiagFunc, rec(
    checkParams := (self, params) >> Checked(Length(params)=2,
        IsPosIntSym(params[1]), IsIntSym(params[2]), params),
    lambda := self >> let(i:=Ind(), Lambda(i,
        omega(self.params[1], self.params[2]*i)).setRange(TComplex)),
    range := self >> TComplex,
    domain := self >> TInt
));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**
  Non-Terminals (Transforms) and rule trees

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Spiral Non-Terminals (Transforms)

Definitions

```
t1 := DFT(4);            # complex DFT of size 4
t2 := MDDFT([4,4]);      # 2D DFT
t3 := DFT(5);            # non 2-power DFT
Import(dct_dst);         # load DCT/DST package
t4 := DCT3(8);           # cosine transform of type 3, size 8
Import(filtering);       # load package filtering
t5 := Filt(4, [1,2,3,4]);     # FIR filter with constant taps
Import(wht);             # load Walsh-Hadamard Transform
t6 := WHT(3);            # WHT of size 8
```

Operations on functions

```
DoForAll([t1,t2,t3,t4,t5,t6], # print them all as matrices
    t->Print(pm(t), "\n"));
t1.terminate();                 # translate into matrix
t4.transpose();                 # transposed transform
t1.conjTranspose();             # conjugated transposed transform
t3.inverse();                   # inverse transform transform
t2.dims();                      # transforms have a size

SpiralDefaults.breakdownRules;# all transforms known to the system
```

Rule Trees

Expand Non-Terminals

```
opts := SpiralDefaults;
t1 := DFT(4);                          # complex DFT of size 4
rt1 := RandomRuleTree(t1, opts);       # create a random rule tree
t2 := DFT(80);                         # complex DFT of size 80
rt2 := RandomRuleTree(t2, opts);       # create a random rule tree
```

Exploring a Rule Tree

```
rt1.node;                              # this node
rt1.rule;                              # rule applied at node
rt1.transposed;                        # rule applied transposed ?
rt1.children;                          # a level down in the tree
rt1.children[1];                       # first child node
rt1.children[1].node;                  # same as root node
rt1.children[1].rule;
rt1.children[1].children;
rt1.children[1].transposed;
rt1.children[2];                       # second child node
rt1.children[2].node;                  # again tree node structure
rt1.children[2].rule;
rt1.children[2].children;
rt1.children[2].transposed;
```

Non-Terminal Example: DFT

Definition

```
# In spiral-core\namespaces\spiral\transforms\dft\dft.gi
Class(DFT, DFT_NonTerm, rec(
    transpose     := self >> DFT(self.params[1],
                        self.params[2]).withTags(self.getTags()),
    conjTranspose := self >> DFT(self.params[1],
                        -self.params[2]).withTags(self.getTags()),
    inverse := self >> self.conjTranspose(),
    omega4pow := (r,c) -> 4*r*c,
));
```

Base Class

```
Class(DFT_NonTerm, TaggedNonTerminal, rec(
    abbrevs := [(n) -> Checked(IsPosIntSym(n), [_unwrap(n), 1]),...],
    hashAs := ...,
    dims := ...,
    terminate := ...,
    isReal := ...,
    SmallRandom := () -> Random([2..16]),
    LargeRandom := () -> 2 ^ Random([6..15]),
    normalizedArithCost := ...
    TType := T_Complex(TUnknown)
));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**
  SPL

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

SPL: Matrices and Symbolic Matrices

Important examples: SPL objects

```
s1 := I(4);                      # Identity matrix
s2 := F(2);                      # Butterfly matrix
s3 := L(8, 2);                   # Stride permutation matrix
s4 := Mat([[1,2],[3,4]]);        # GAP Matrix as SPL object
RowVec(4); ColVec(4);            # row and column vectors
O(4); O(3, 4);                   # zero matrix, square and rectangular
```

Important examples: SPL Operations

```
s2 * s4; Compose(s2, s4);        # product of SPL
DirectSum(s1, s2);               # matrix direct sum
Tensor(s1, s2);                  # Kronecker product of matrices
HStack(s2, s4);                  # [[s2,s4]]
VStack(s2, s4);                  # [[s2],[s4]]
```

Operations for SPL objects

```
pm(s1);                          # print as matrix
MatSPL(s2);                      # convert to GAP matrix
s3.transpose();                  # symbolic transposition
# List all SPL objects
List(Filtered(Dir(spiral.spl), o->IsSPL(spiral.spl.(o))),
    e->spiral.spl.(e));
```

Still SPL, But Towards Σ-SPL

Permutations

```
s1 := L(8,4);              # The stride permutation...
MatSPL(s1);                # ...is a matrix...
s1.lambda();               # ...and a symbolic function
L(8, 4) * L(8, 2);         # == I(8)
Tensor(I(2), L(4,2)) * L(8,2);        # digit perm needs expression

# other permutation matrices/functions
J(4); Z(4,1); CRT(4,5); RR(13,3,2);
Tensor(J(4), Z(4, 1));
```

Diagonals

```
d:= Diag(fConst(4,1.1));                  # Diagonal matrix
d.element;
e:= RCDiag(FList(TReal, [1..16]));    # RC(Diag(...))
e.element;
f := DiagCpxSplit(FList(TReal, [1..16]));
f.element;

# diagonal matrix with dependency on free variable
i := Ind(2);
Diag(fCompose(FList(TReal, [1..4]), fTensor(fId(2), fBase(i))));
```

More SPL Operators

More Operators

```
s1 := DFT(4).terminate();      # Get the complex DFT(4) matrix
s2 := RC(s1);                  # convert it to a real 8x8 matrix
s3 := COND(Ind(), I(2), F(2));     # conditional matrix
s4:= Tensor(DFT(4), I(4));         # transforms are SPL objects
MatSPL(DFT(4)) * [1..4];     # and support SPL and Matrix operations
ConjLR(Tensor(I(2), F(2)), L(4,2), L(4,2));  # classical identity

RowDirectSum(1, F(2), J(2));  # overlapped direct sum
RowTensor(5, 1, F(2));        # overlapped tensor
ColDirectSum(1, F(2), J(2));  # overlapped direct sum
ColTensor(5, 1, F(2));        # overlapped tensor
```

Iterative Operators

```
i := Ind(4);
s5 := IterDirectSum(i, F(2));        # iterative direct sum
s6 := IterDirectSum(i, Mat([[i+1, 2*i],[-3*i, 4*i+5]]));
MatSPL(s6);

s7 := IterHStack(i, F(2));            # iterative HStack
s8 := IterVStack(i, F(2));            # iterative VStack
```

Defining SPL Objects

SPL Matrices

```
# spiral-core\namespaces\spiral\spl\symbols.gi
Class(F, Sym, rec(
    def := size -> Checked(IsPosInt(size),
    Cond(size = 1, Mat([[1]]),
        size = 2, Mat([[1,1], [1,-1]]),
        Mat(Global.DFT(size)))),

    isReal     := self >> self.params[1] <= 2,
    isPermutation := False,
    transpose := self >> self,
    conjTranspose := self >> self,
    inverse := self >> let(n:=self.params[1],
        Cond(n=1, self, n=2, 1/2*F(2),
        Error("Inverse not supported"))),
    toAMat     := self >> DFTAMat(self.params[1]),
    printlatex := (self) >> Print(" F_{", self.params[1], "} ")
));
```

Example

```
s := F(2);
MatSPL(s);
```

Defining SPL Objects

SPL Operator

```
# spiral-core\namespaces\spiral\spl\Conjugate.gi
Class(Conjugate, BaseOperation, rec(
    new := (self, spl, conj) >> Checked(IsSPL(spl), IsSPL(conj),
        When(Dimensions(conj) = [1,1], spl,
            SPL(WithBases( self,
                    rec( _children := [spl, conj],
                        dimensions := spl.dimensions ))))),
    isPermutation := False,
    dims := self >> self.dimensions,
    toAMat := self >>
        ConjugateAMat(AMatSPL(self._children[1]),
            AMatSPL(self._children[2])),
    print := (self, i, is) >>
        Print("(", self.child(1).print(i, is), ") ^ (",
        self.child(2).print(i+is, is), ")", self.printA()),
    transpose := self >> Inherit(self, rec(_children := [
        TransposedSPL(self._children[1]), self._children[2] ],
    dimensions := Reversed(self.dimensions))),
    arithmeticCost := (self, costMul, costAddMul) >>
        self._children[1].arithmeticCost(costMul, costAddMul)
));
```

Defining SPL Objects

SPL Permutation

```
# spiral-core\namespaces\spiral\spl\perms.gi
Class(L, PermClass, rec(
    def := (n,str) -> Checked(IsPosIntSym(n), IsPosIntSym(str),
        (not (IsInt(n) and IsInt(str)) or n mod str = 0), rec()),

    domain := self >> self.params[1],
    range  := self >> self.params[1],

    lambda := self >> let(
        n := self.params[1], str := self.params[2], i := Ind(n),
        Lambda(i, idiv(i, n/str) + str * imod(i, n/str))),

    transpose := self >> self.__bases__[1](self.params[1],
        self.params[1] / self.params[2]),
    isSymmetric := self >> (self.params[1] = self.params[2]^2) or
        (self.params[2] = 1) or (self.params[2] = self.params[1]),
));
```

Transforms, RuleTrees, and SPL Formulas

From Transform to SPL

```
# create the objects and lower them
t := DFT(4);
rt := RandomRuleTree(t, SpiralDefaults);
s := SPLRuleTree(rt);
```

Conversions Transform/SPL/GAP Matrix

```
t.terminate();                  # transform -> SPL
tm := MatSPL(t);                # transform -> GAP matrix
sm := MatSPL(s);                # SPL formula -> GAP matrix
```

Symbolic Correctness

```
tm = sm;                        # same, as GAP matrices
InfinityNormMat(tm - sm);       # computes the matrix norm

t1 := DFT(13);                  # for this we lose symbolic equivalency
rt1 := RandomRuleTree(t1, SpiralDefaults);
s1 := SPLRuleTree(rt1);         # size 13 triggers Rader
tm1 := MatSPL(t1);              # this is fully symbolic, but
sm1 := MatSPL(s1);              # Rader requires the FFT
tm1 = sm1;                      # thus floating-point matrix
InfinityNormMat(tm1 - sm1);     # but equivalent wrt. floating-point
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**
  Σ-SPL and loops

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Σ-SPL: Gather and Scatter

Gather and Scatter matrices

```
g1 := Gath(fId(2));                         # Gath(fId(.)) = I(.)
g2 := Gath(fBase(4, 0));                     # Gath(fBase(.,.)) = base vec
g3 := Gath(fTensor(fBase(4, 0), fId(2)));    # standard pattern

s1 := Scat(fId(2));                          # Scat(fId(.)) = I(.)
s2 := Scat(fBase(4, 2));                      # Scat(fBase(.,.)) = base vec
s3 := Scat(fTensor(fId(2), fBase(4, 3)));    # standard pattern
```

Scatter/Kernel/Gather Pattern

```
A := F(2); j := 0;
# iteration j of Tensor(I(4), F(2))
sag1 := Scat(fTensor(fBase(4, j), fId(2))) * A *
        Gath(fTensor(fBase(4, j), fId(2)));

# iteration j of Tensor(F(2), I(4))
sag2 := Scat(fTensor(fId(2), fBase(4, j))) * A *
        Gath(fTensor(fId(2), fBase(4, j)));

# iteration j of Tensor(I(4), F(2)) * L(8, 4)
sag3 := Scat(fTensor(fBase(4, j), fId(2))) * A *
        Gath(fTensor(fId(2), fBase(4, j)));
pm(sag1); pm(sag2); pm(sag3);
```

Σ-SPL: Gather, Scatter, and ISum

Tensor Product and Gath/Scat/ISum

```
A := F(2);
j := Ind(4);
# Sigma-SPL for Tensor(I(4), F(2))
sag1 := Scat(fTensor(fBase(j), fId(2))) * A *
        Gath(fTensor(fBase(j), fId(2)));
s1 := ISum(j, sag1);
MatSPL(s1) = MatSPL(Tensor(I(4), F(2)));
```

Other Tensor Patterns

```
# Sigma-SPL for Tensor(F(2), I(4))
s2 := ISum(j, Scat(fTensor(fId(2), fBase(j))) * A *
        Gath(fTensor(fId(2), fBase(j))));
MatSPL(s2) = MatSPL(Tensor(F(2), I(4)));


# Sigma-SPL for Tensor(I(4), F(2)) * L(8, 4)
s3 := ISum(j, Scat(fTensor(fBase(j), fId(2))) * A *
        Gath(fTensor(fId(2), fBase(j))));
MatSPL(s3) = MatSPL(Tensor(I(4), F(2)) * L(8, 4));


# Direct sum
ISum(j, Scat(fTensor(fBase(j), fId(2))) * Mat([[j,-j],[j, j]]) *
        Gath(fTensor(fBase(j), fId(2))));
```

Σ-SPL: Advanced Loops

Tensor Product and Gath/Scat/ISum

```
A := F(2);
j := Ind(4);
k := Ind(2);

# Sigma-SPL for Tensor(I(4), F(2), I(2))
sag := Scat(fTensor(fBase(j), fId(2), fBase(k))) * A *
       Gath(fTensor(fBase(j), fId(2), fBase(k)));
s := ISum(k, ISum(j, sag));
MatSPL(s) = MatSPL(Tensor(I(4), F(2), I(2)));
```

More complex example

```
i := Ind(8);
j := Ind(4);
k := Ind(2);
A := Mat([[j+1,-2*j],[j*k, j+1]]); B := F(2);

s := ISum(k, ISum(j, Scat(fTensor(fBase(j), fBase(k), fId(2))) * A *
                     Gath(fTensor(fId(2), fBase(k), fBase(j))))) *
     ISum(i, Scat(fTensor(fBase(i), fId(2))) * B
          * Gath(fTensor(J(2), fCompose(Z(8,3), fBase(i)))));
MatSPL(s);
```

Σ-SPL: Gath/Scat, Diag and Perms

Gather Functions

```
# use Lambda functions directly in Gath/Scat
i := Ind(8);
f1 := Lambda(i, imod(5*i+7, 32)).setRange(32);
g := Gath(f1);

# Use indirection tables in Gath/Scat. By default not supported
f2 := CopyFields(FData(List([0..7], j->V(Mod(5*j+7, 32)))),
                 rec(range := self >> 32));
s := Scat(f2);
```

Diagonals

```
# the true Twiddle diagonal in DFT(8)
d := Diag(fPrecompute(fCompose(dOmega(8, 1),
         diagTensor(dLin(V(4), 1, 0, TInt), dLin(2, 1, 0, TInt)))));
MatSPL(d);
e:= RCDiag(fCompose(FData(List([1..32], u->Value(TReal, u))),
         fTensor(fId(4), fBase(i))));
# print out the function values
e.element.tolist();
# access the indirection table
e.element.children()[1].var.value;
```

Defining Σ-SPL Objects

Gather

```
# spiral-core\namespaces\spiral\spl\sums.gi
Class(Gath, SumsBase, BaseMat, rec(
    rChildren := self >> [self.func],
    rSetChild := rSetChildFields("func"),
    new := (self, func) >> SPL(WithBases(self, rec(func :=
        Checked(IsFunction(func) or IsFuncExp(func), func)))).setDims(),
    dims := self >> [self.func.domain(), self.func.range()],
    sums := self >> self,
    area := self >> Sum(Flat([self.func.domain()])),
    isReal := self >> true,
    transpose := self >> Scat(self.func),
    conjTranspose := self >> self.transpose(),
    inverse := self >> self.transpose(),
    toAMat := self >> let(
      n := EvalScalar(self.func.domain()),
      N := EvalScalar(self.func.range()),
      func := self.func.lambda(),
      AMatMat(List([0..n-1], row -> let(idx:=evalScalar(func.at(row)),
          Cond(idx _is funcExp, When(idx.args[1]=0, Replicate(N, 0),
                     Error("... ")),
              BasisVec(N, idx)))))),
));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**
  End-to-end flow

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

End to End from DFT(8) -> C Code

From Transform to code -- stepwise

```
n := 8; k := -1;                    # transform parameters
opts := SpiralDefaults;             # default options
opts.useDeref := false;             # prefer array[] over *(deref)
t := DFT(n, k);                     # transform
rt := RandomRuleTree(t, opts);          # get rule tree
spl := SPLRuleTree(rt);             # Debug: SPL formula
ss1 := spl.sums();      # Debug: SPL->Sigma-SPL w/o optimization
ss := SumsRuleTree(rt, opts); # Correct: from rt -> Sigma-SPL
c1 := CodeSums(ss, opts);           # Debug: Sigma-SPL->code
c := CodeRuleTree(rt, opts);  # Correct: rt-> code in one shot
PrintCode("dft8", c, opts);    # final code
```

Correctness checks

```
tm := MatSPL(t);                    # symbolic complex cyclotomic matrix
tmr := MatSPL(RC(t));               # symbolic real cyclotomic matrix
splm := MatSPL(spl);                # symbolic complex cyclotomic matrix
tmr := MatSPL(RC(t));               # symbolic real cyclotomic matrix
ssm := MatSPL(ss);                  # symbolic double-precision matrix
cm := CMatrix(c, opts);             # symbolic double-precision matrix
tm = splm;                          # symbolically equivalent
InfinityNormMat(tmr - ssm);         # only equivalent up to rounding error
InfinityNormMat(tmr - cm);          # only equivalent up to rounding error
```

More Examples

From Transform to code -- stepwise

```
n := 1024; k := -1;              # transform parameters
opts := SpiralDefaults;          # default options
opts.globalUnrolling := 16;      # set smaller unrolling
t := DFT(n, k);                  # transform
best := DP(t, rec(), opts);      # run search
rt := best[1].ruletree;
c := CodeRuleTree(rt, opts);     # Correct: rt-> code in one shot
PrintCode("dft"::StringInt(n), c, opts);      # final code
```

Other Examples

```
Import(dct_dst, realdft);        # load DCT/DST and Real DFT package
opts := SpiralDefaults;          # default options
t1 := DFT(31);           # a larger prime size
t2 := DCT3(32);          # a larger cosine transform of type 3
t3 := PRDFT(17);         # Real DFT in the "pack" format
t4 := PrunedDFT(128, 16, [0,1,5,6,7]);

ts := [t1, t2, t3, t4];
rts := List(ts, tt->RandomRuleTree(tt, opts));
cs := List(rts, rr->CodeRuleTree(rr,
        CopyFields(SpiralDefaults, rec(globalUnrolling := 64))));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**
  Ruletree system and search

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Spiral
Software/Hardware Generation for Performance

SpiralGen
Accelerating Innovation in Computing

Breakdown Rules: Base Rule

Definition

```
# In spiral-core\namespaces\spiral\transforms\dft\dft_rules.gi
 DFT_Base := rec(
    forTransposition := false,
    applicable       := nt -> nt.params[1] = 2 and not nt.hasTags(),
    apply            := (nt, C, cnt) -> F(2)
)
```

Twiddle function for DFT

```
Tw1 := (n,d,k) -> Checked(
    IsPosIntSym(n), IsPosIntSym(d), IsIntSym(k),
    fCompose(dOmega(n,k),
        diagTensor(dLin(div(n,d), 1, 0, TInt),
            dLin(d, 1, 0, TInt))));
```

Rule methods

```
PrintActiveRules(DFT);          # rules for DFT currently active
DFT_Base.switch;                # filed in rule to determine active
t := DFT(2);
DFT_Base.applicable(t);         # is the rule applicable
DFT_Base.children(t);           # all possible Algorithmic choices
DFT_Base.apply(t, [], []);      # t->spl for a particular choice
```

Breakdown Rules: Cooley-Tukey Rule

Definition

```
# In spiral-core\namespaces\spiral\transforms\dft\dft_rules.gi
DFT_CT := rec(
    maxSize          := false,
    forcePrimeFactor := false,
    applicable := (self, nt) >> nt.params[1] > 2
        and not nt.hasTags()
        and (self.maxSize=false or nt.params[1] <= self.maxSize)
        and not IsPrime(nt.params[1])
        and When(self.forcePrimeFactor, not
                 DFT_GoodThomas.applicable(nt), true),
    children  := nt -> Map2(DivisorPairs(nt.params[1]),
            (m,n) -> [ DFT(m, nt.params[2] mod m),
                       DFT(n, nt.params[2] mod n) ]),
    apply := (nt, C, cnt) -> let(mn := nt.params[1],
                m := Rows(C[1]), n := Rows(C[2]),
            Tensor(C[1], I(n)) *
            Diag(fPrecompute(Tw1(mn, n, nt.params[2]))) *
            Tensor(I(m), C[2]) *
            L(mn, m)
        )
    )
```

Breakdown Rules: Cooley-Tukey Rule

Applicability: Cooley Tukey requires non-prime size

```
t  := DFT(2);
t1 := DFT(4);
t2 := DFT(8);
t3 := DFT(20);

DFT_CT.applicable(t);          # see for which sized DFT_CT
DFT_CT.applicable(DFT(5));     # is applicable
DFT_CT.applicable(t1);
DFT_CT.applicable(t2);
DFT_CT.applicable(t3);
```

Children: algorithmic choices

```
c1 := DFT_CT.children(t2);     # enumerate all algorithmic choices
c2 := DFT_CT.children(t2);
c3 := DFT_CT.children(t3);
```

Expand DFT(8) by hand

```
s  := DFT_Base.apply(t, [], []);       # expand DFT(2) -> F(2)
s1 := DFT_CT.apply(t1, [s, s], [t, t]);      # DFT(4) -> SPL
s2 := DFT_CT.apply(t2, [s1, s], [t1, t]);    # DFT(8) -> SPL
MatSPL(t2) = MatSPL(s2);
```

Ruletrees and SPL Revisited

From Transform to SPL Formula

```
n := 8; k := -1;                          # transform parameters
opts := CopyFields(SpiralDefaults,        # local configuration
    rec(breakdownRules := rec(
        DFT := [DFT_Base,
        CopyFields(DFT_CT, rec(maxSize := 20))])));
t := DFT(n, k);                           # transform
rt := RandomRuleTree(t, opts);            # get rule tree
spl := SPLRuleTree(rt);                   # SPL formula
```

Impact of configuration

```
PrintActiveRules(DFT);
opts.breakdownRules.DFT;
DFT_CT.maxSize;                           # global configuration unchanged
ct := Filtered(opts.breakdownRules.DFT, i->i.name =  DFT_CT.name)[1];
ct.maxSize;                               # access local configuration
t2 := DFT(21);                            # works with global but not local opts
rt := RandomRuleTree(t2, SpiralDefaults);
rt2 := RandomRuleTree(t2, opts);
FindUnexpandableNonterminal(t2, opts);         # Where do we fail?
ct.maxSize := false;                           # remove guard in DFT_CT
rt2 := RandomRuleTree(t2, opts);               # try again
FindUnexpandableNonterminal(t2, opts);         # Where do we fail now?
```

Search: Dynamic Programming

Standard dynamic programming

```
n := 15; k := -1;                # transform parameters
opts := SpiralDefaults;          # default options
opts.globalUnrolling := 16;      # set smaller unrolling
t := DFT(n, k);                  # transform
best := DP(t, rec(), opts);      # run search
rt := best[1].ruletree;          # get best rule tree
```

Hashing and custom measure functions

```
dpopts := rec(verbosity := 5, hashTable := HashTableDP());
dpopts.measureFunction := (rt, opts) ->
    let(c:= CodeRuleTree(rt, opts),   # generate code
        Length(Filtered(                # count flops in code
            Collect(c, @(1,[add, sub, mul, neg])), i->i.t=TReal)));
best := DP(t, dpopts, opts);  # run search with flop minimization
#look what's in the hash table
HashLookup(dpopts.hashTable, DFT(5, 1))[1].ruletree;
HashLookup(dpopts.hashTable, DFT(5,-1));

# now time the tree found through flop minimization
rt1 := HashLookup(dpopts.hashTable, DFT(15, 1))[1].ruletree;
DPMeasureRuleTree(rt1, opts);
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**
  Rewriting system

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Pattern Matching

Collect

```
opts := SpiralDefaults;
s := SumsRuleTree(RandomRuleTree(DFT(8), opts), opts);
c := CodeSums(s, opts);


Collect(s, Scat);                # get list of scatter operations
Set(Collect(s, Value));          # get all unique values
```

Simple Patterns

```
Collect(c, @(1, [add, sub, neg, mul]));      # get all arith ops...
Collect(c, @(1, [add, sub, neg, mul], e->e.t=TReal)); #...on reals

List(Collect(s, @(1, ISum)), e->e.var);      # all loop variables
Set(Collect(s, @@(1, Value,    # all values inside Blk objects
    (e, cx)->IsBound(cx.Blk) and Length(cx.Blk) > 0)));
```

Subtree patterns

```
Collect(c, [deref, add, sub]);
Collect(c, [mul, @(1), sub]);
Collect(c, [mul, Value, ...]);
Collect(c, [mul, @(1), [sub, deref, @(2)]]);
Collect(c, [mul, @(1), [sub, @(2, deref, e->X in e.free()), @(3)]]);
```

Substitutions

SubstTopDown/SubstBottomUp

```
opts := SpiralDefaults;
c := CodeSums(SumsRuleTree(RandomRuleTree(DFT(8), opts), opts), opts);

# Ordered substitution: traversal order can matter greatly
SubstTopDown(Copy(c), @(1, Value, e->e.v=1), e->V(25));
SubstBottomUp(Copy(c), @(1, Value, e->e.v=1), e->V(-25));
```

Variable substitutions

```
vars := Collect(c, @(1, var, e->e.t=TReal)); # all the real variables
SubstVars(Copy(c), rec((vars[1].id) := V(1.1)));     # substitute one

# record of assignment of consecutive numbers to all real variables
substrec := FoldR(Zip2(vars, [1..Length(vars)]),
    (a,b) -> CopyFields(a, rec((b[1].id) := V(b[2]))), rec());
SubstVars(Copy(c), substrec); # substitute them

# loop unrolling example
i := Ind(4);
c2 := loop(i, 4, assign(nth(X, i), i));       # loop to be unrolled
chain(List(c2.range,  # chain of partially evaluated loop iterations
    i->SubstVars(Copy(c2.cmd), rec((c2.var.id) := V(i)))));
```

Rules

Simple Rules

```
Rule([neg, [neg, @1]], e -> @1.val);
Rule([add, Value, Value],
    e->Value.new(e.args[1].t, e.args[1].v + e.args[2].v));
Rule([im, [conj, @(1)]], x->-im(@(1).val));
Rule([IF, @(1), skip, skip], e -> skip());

Rule([RC, @(1, Compose)], e -> Compose(List(@(1).val.children(), RC)));
Rule([RC, @(1, Gath)], e -> Gath(fTensor(@(1).val.func, fId(2))));

Rule([Tensor, ..., @(1,O), ...], e -> O(Rows(e), Cols(e)));
```

Complex Rules

```
_v0none := @(0).target([ Value, noneExp ]).cond(
    (e) -> Cond(ObjId(e) = noneExp, true, isValueZero(e)));
_0noneOrZero :=(t) -> When(
    ObjId(@(0).val) = noneExp, noneExp(t), t.zero());
Rule([mul, ..., _v0none, ...], e -> _0noneOrZero(e.t));
Rule([@@(0,mul,(e,cx)->IsBound(cx.nth) and cx.nth<>[]), @(1), @(2,add)],
    e -> ApplyFunc(add, List(@(2).val.args, a->@(1).val * a)));
Rule( [im, [mul, [cxpack, @(1), @(2)], [conj, [cxpack,
    @(3).cond(x->x=@(1).val), @(4).cond(x->x=@(2).val)]]]],
    e -> e.t.zero() );
```

Associative Rules

Simple Rules

```
ARule(add, [ @(1,add) ], e -> @1.val.args);
ARule(fTensor,  [@(1, fTensor) ], e -> @(1).val.children());
ARule(fCompose, [@(1), fId ], e -> [@(1).val]);


ARule(Compose, [ @(1, Prm), @(2, Prm) ],
    e -> [ Prm(fCompose(@(2).val.func, @(1).val.func)) ])
ARule(Compose, [ @(1, Gath), @(2, [Gath, Prm]) ],
    e -> [ Gath(fCompose(@(2).val.func, @(1).val.func)) ]);
```

Complex Rules

```
ARule(leq, [@(1, Value, x->x.v<=0), [@(0,mul), @(2, Value, x->x.v>0),
    @(3,var,IsLoopIndex)]], e -> [@(0).val]);


ARule( Compose, [ @(1, [Prm, Scat, ScatAcc, Conj, ConjL, ConjR, ConjLR]),
    @(2, [RecursStep, Grp, BB, SUM, ISum, Data, COND]) ],
    e -> [ CopyFields(@(2).val, rec(
            _children :=  List(@(2).val._children, c -> @(1).val * c),
            dimensions := [Rows(@(1).val), Cols(@(2).val)] )) ]);


ARule(fCompose, [ @(1, L), [ @(3, fTensor),
    @(2).cond(e->range(e) = @(1).val.params[2] and domain(e)=1), ... ] ],
    e->[ fTensor(Copy(Drop(@(3).val.children(), 1)), Copy(@(2).val)) ] );
```

Rule Sets

Define a Rule Set

```
# spiral-core\namespaces\spiral\code\sreduce.gi
Class(RulesStrengthReduce, RuleSet);
RewriteRules(RulesStrengthReduce, rec(
    leq_single := Rule([leq, @(1)], e-> V_true),
    add_assoc  := ARule(add, [ @(1,add) ], e -> @1.val.args),
# hundreds of rules
));
```

Add Rules to Existing Rule Set

```
# somewhere else in the source code
RewriteRules(RulesStrengthReduce, rec(       # add more rules
    logic_single := Rule([@(1, [logic_and, logic_or]), @1], e->@1.val)
));
```

Using Rule Sets

```
RulesStrengthReduce.rules.leq_single;
opts := SpiralDefaults;
s := SPLRuleTree(RandomRuleTree(DFT(8), opts)).sums();
s := Rewrite(s, RulesSums, opts);
s := Rewrite(s, RulesDiag, opts);
s := RulesDiagStandalone(s);
```

Rule Strategies

Define a Rule Strategy

```
# spiral-core\namespaces\spiral\code\sreduce.gi
LibStrategy := [ StandardSumsRules, HfuncSumsRules ];
```

Combining Rule Sets

```
StandardSumsRules := MergedRuleSet(
    RulesSums, RulesFuncSimp, RulesDiag, RulesDiagStandalone,
    RulesStrengthReduce, RulesRC, RulesII, OLRules
);
```

Use of Rule Strategies

```
SpiralDefaults.formulaStrategies.sigmaSpl;
SpiralDefaults.formulaStrategies.rc;
SReduce := (c,opts) ->          # handy shortcut
    ApplyStrategy(c, [RulesStrengthReduce], BUA, opts);

opts := SpiralDefaults;
s := SumsRuleTree(RandomRuleTree(DFT(4), opts), opts);
c := DefaultCodegen(s, Y, X, opts);
c := SubstTopDown(c, @(1, loop), e->e.unroll());
Collect(c, mul);
c := SReduce(c, opts);
Collect(c, mul);
```

Rewriting: General Mechanics

Interface for rewriting

```
# spiral-core\namespaces\spiral\code\ir.gi
Class(deref, nth, rec(
    __call__ := (self, loc) >> Inherited(loc, TInt.value(0)),
    rChildren := self >> [self.loc],
    rSetChild := rSetChildFields("loc"),
));
deref.from_rChildren;
```

Unifying interface across all rewritable objects

```
c := deref(X);              # code objects
c.rChildren();             # get rewritable fields
c.rSetChild(1, Y);         # change a rewritable field

DFT.rChildren;             # Transform level
RandomRuleTree(DFT(4), SpiralDefaults).rChildren;    # ruletree level
F.rChildren;               # SPL level
L.rChildren;               # Permutations
Gath.rChildren;            # Sigma-SPL
Lambda.rChildren;          # Lambda function
fId.rChildren;             # symbolic functions
add.rChildren;             # expressions
T_Real.rChildren;          # data types
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**
  Translators and visitor pattern

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Implementing Recursive Descent: Visitors

Simple example

```
# spiral-core\namespaces\spiral\rewrite\visitor.gi
Class(LispGen, Visitor, rec(
    add := (self, o) >> Print("(+ ", self(o.args[1]), " ",
        self(o.args[2]), ")"),
    mul := (self, o) >> Print("(* ", self(o.args[1]), " ",
        self(o.args[2]), ")"),
    sub := (self, o) >> Print("(- ", self(o.args[1]), " ",
        self(o.args[2]), ")"),
    var := (self, o) >> Print("(var ", o.id, ")"),
    Value := (self, o) >> Print("(value ", o.v, ")")
));
LispGen(4*X+2);
```

Visitors used in standard translation flow

```
opts := SpiralDefaults;
opts.sumsgen;
DefaultSumsGen;
opts.codegen;
DefaultCodegen;
opts.unparser;
CUnparser;
```

SumsGen

The DefaultSumsGen Visitor

```
# spiral-core\namespaces\spiral\sigma\sumsgen.gi
# all the fields
Filtered(RecFields(DefaultSumsGen), i->not IsSystemRecField(i));
Print(DefaultSumsGen.__call__);

# the recursive definitions needed for DFT
DefaultSumsGen.Compose;
Print(DefaultSumsGen.Tensor);
DefaultSumsGen.I;
DefaultSumsGen.F;
DefaultSumsGen.Diag;
DefaultSumsGen.L;
```

Visitors used in standard translation flow

```
opts := SpiralDefaults;
s := SPLRuleTree(RandomRuleTree(DFT(8), opts));
SumsSPL(s, opts);
opts.sumsgen(s, opts);

# legacy and backwards compatibility framework
F(2).sums();
Tensor(F(2), I(2)).sums();
```

CodeGen

The DefaultCodegen Visitor

```
# spiral-core\namespaces\spiral\compiler\codegen.gi
# all the fields
Filtered(RecFields(DefaultCodegen), i->not IsSystemRecField(i));
Print(DefaultCodegen.__call__);


# Some of the fields
Print(DefaultCodegen.Formula);
Print(DefaultCodegen.Compose);
DefaultCodegen.ISum;
Print(DefaultCodegen.Gath);
Print(DefaultCodegen.Scat);
DefaultCodegen.Diag;
```

Visitors used in standard translation flow

```
opts := SpiralDefaults;
s := SumsRuleTree(RandomRuleTree(DFT(8), opts), opts);
# only translate Sigma-SPL to icode
opts.codegen(s, Y, X, opts);
# also invoke the basic block compiler
opts.codegen(Formula(s), Y, X, opts);
```

C Pretty Printer

The CUnparser Visitor

```
# spiral-core\namespaces\spiral\compiler\unparse.gi
# all the fields
Filtered(RecFields(CUnparser), i->not IsSystemRecField(i));
Filtered(RecFields(CUnparserBase), i->not IsSystemRecField(i));
Print(CUnparser.gen);

# Some of the fields
Print(CUnparser.loop);
CUnparser.deref;
CUnparser.add;
CUnparser.Value;
Print(CUnparser.decl);
CUnparser.chain;
```

Visitors used in standard translation flow

```
opts := SpiralDefaults;
c := CodeRuleTree(RandomRuleTree(DFT(8), opts), opts);
# Print full header etc.
PrintCode("dft8", c, opts);
# unparser needs opts as context
Unparse(c.cmds[1].cmds[2].cmd,
    CopyFields(CUnparser, rec(opts:=opts)), 0, 1);
```

Organization

■ **Installation**

■ **GAP3 language**

■ **Spiral objects and data types**

■ **Spiral infrastructure**
  Basic block compiler

■ **Special hardware: SMP/OpenMP**

■ **Special hardware: SIMD/AVX**

■ **Benchmark/production code infrastructure**

Compiling from Σ-SPL to icode

Top-level flow

```
opts := SpiralDefaults;
s := SumsRuleTree(RandomRuleTree(DFT(4), opts), opts);
c := CodeSums(s, opts);
```

Basic Block Compilation

```
# the actual code generator is a configuration option
opts.codegen;
# What happens in CodeSums
DefaultCodegen(Formula(s), Y, X, opts);

# without Formula() the basic block compiler is not run
c := DefaultCodegen(s, Y, X, opts);
# invoke the basic block compiler
Compile(c, opts);

# Compile calls a number of compile strategies
opts.compileStrategy;
for i  in [ 1 .. Length(opts.compileStrategy) ] do
    c := let(stage := opts.compileStrategy[i],
        When(IsCallableN(stage, 2), stage(c, opts), stage(c)));
od;
c;
```

Basic Block Compiler

Top-level flow

```
opts := SpiralDefaults;
s := SumsRuleTree(RandomRuleTree(DFT(8), opts), opts);
c := DefaultCodegen(s, Y, X, opts);
Compile(c, opts);
```

Basic Block Compilation, Stage by Stage

```
c := Compile.pullDataDeclsRefs(c);
c := Compile.fastScalarize(c);
c := UnrollCode(c);
c := FlattenCode(c);
c := UntangleChain(c);
c := CopyPropagate.initial(c, opts);
c := HashConsts(c, opts);
c := MarkDefUse(c);
c := BinSplit(c, opts);
c := MarkDefUse(c);
c := CopyPropagate(c, opts);
c := BinSplit(c, opts);
c := FixValueTypes(c);
c := Compile.declareVars(c);
PrintCode("dft8", c, opts);
```

A Closer Look at Compiler Stages

Implementation of important stages

```
Print(Compile.pullDataDeclsRefs);
Print(Compile.fastScalarize);
UnrollCode;
FlattenCode;
UntangleChain;
Print(CopyPropagate.copyProp);
Print(CSE.__call__);
Print(HashConsts);
Print(_MarkDefUse);
Print(BinSplit.__call__);
FixValueTypes;
Compile.declareVars;
```

Organization

■ **Installation**

■ **GAP3 language**

■ **Spiral objects and data types**

■ **Spiral infrastructure**
   Profiler

■ **Special hardware: SMP/OpenMP**

■ **Special hardware: SIMD/AVX**

■ **Benchmark/production code infrastructure**

Profiler

Top-level flow

```
opts := SpiralDefaults;
c := CodeRuleTree(RandomRuleTree(DFT(8), opts), opts);
PrintCode("dft8", c, opts);
CMeasure(c, opts);                 # measure the runtime
CMatrix(c, opts);                  # construct the transform matrix from c
```

Inspect Profiles

```
opts.profile;
default_profiles;
spd := GetEnv("SPIRAL_DIR");
Exec("dir ":: spd::"\\profiler\\targets");
Exec("dir ":: spd::"\\profiler\\targets\\win-x86-vcc");
Exec("type ":: spd::"\\profiler\\targets\\win-x86-vcc\\Makefile");
```

Look at the disk contents

```
# see in which drive we are. Usually C: or D:
Exec("cd");
# if no outdir is bound in opts this is the default temp path
IsBound(opts.outdir);
Exec("dir \\tmp\\"::StringInt(GetPid()));
Exec("type \\tmp\\"::StringInt(GetPid())::"\\testcode.c");
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**
  Tags and tSPL

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Targeting Advanced Hardware

Simple Example: Multicore/OpenMP

```
opts := IAGlobals.getOpts(rec(dataType := T_Real(64)),
    rec(numproc := 2, api := "OpenMP", OmpMode := "for"));
Add(opts.breakdownRules.TRC, TRC_tag);         # needs some cleanup
opts.tags := opts.tags{[1]};                    # needs some cleanup

t := TRC(DFT(4)).withTags(opts.tags);          # need to add TRC(.)
rt := RandomRuleTree(t, opts);
c := CodeRuleTree(rt, opts);
PrintCode("DFT4_OMP", c, opts);
```

Stepwise code generation

```
opts.tags;                          # what are the tags
spl := SPLRuleTree(rt);             # There are SMP SPL objects
s := SumsRuleTree(rt, opts);        # and a SMP ISum

opts.unparser := OpenMP_Unparser;   # now we use parallel region
PrintCode("DFT4_OMP", c, opts);
```

Tags

Definition

```
# spiral-core\namespaces\spiral\paradigms\smp\sigmaspl.gi
Class(AParSMP, AGenericTag, rec(
    isSMP := true,
    updateParams := meth(self)
        if Length(self.params)=1 then self.params :=
            [self.params[1], threadId()];
        elif Length(self.params)=2 then ;
        else Error("Usage: AParSMP(<num_threads>, [<tid>])");
        fi;
    end
));
```

Use case

```
Import(paradigms.smp);                # not imported by default
tag := AParSMP(2);                    # define a SMP/OpenMP tag
t := TRC(DFT(4)).withTags([tag]);     # tag a transform
pm(t);                                # tagged transforms need to
                                      # be in real arithmetic

MatSPL(DFT(2));                       # TRC(.) lifts RC(.) to the
MatSPL(TRC(DFT(2)));                  # transform level
```

Tagged SPL Non-Terminals

Lift RC to Transform Level

```
# spiral-core\namespaces\spiral\paradigms\common\nonterms.gi
Class(TRC, Tagged_tSPL_Container, rec(
    abbrevs :=  [ (A) -> Checked(IsNonTerminal(A) or IsSPL(A), [A]) ],
    dims := self >> 2*self.params[1].dims(),
    terminate := self >> Mat(MatSPL(RC(self.params[1]))),
    transpose := self >> ObjId(self)(
        self.params[1].conjTranspose()).withTags(self.getTags()),
    conjTranspose := self >> self.transpose(),
  ...
));
```

Lift Compose to Transform Level

```
Class(TCompose, Tagged_tSPL_Container, rec(
    abbrevs :=  [ (l) -> Checked(IsList(l), [l]) ],
    dims := self >> [self.params[1][1].dims()[1],
        self.params[1][Length(self.params[1])].dims()[2]],
    terminate := self >> Compose(
        List(self.params[1], i->i.terminate())),
    transpose := self >> TCompose(Reversed(List(self.params[1],
        i->i.transpose()))).withTags(self.getTags()),
    ...
));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**
  Tagged breakdown rules

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Tag Propagation Rules

Propagate Tag over TRC(.)

```
# spiral-core\namespaces\spiral\paradigms\common\breakdown.gi
TRC_tag := rec(
        forTransposition := false,
        applicable := (self, nt) >> (nt.isTag(1,
            spiral.paradigms.smp.AParSMP) or not nt.hasTags())
            and not nt.hasTag(spiral.paradigms.vector.AVecReg)
            and not nt.hasTag(spiral.paradigms.vector.AVecRegCx),
        children := nt -> [[ nt.params[1].withTags(nt.getTags()) ]],
        apply := (nt, c, cnt) -> RC(c[1])
)
```

Push Tags to Factors of Compose(…)

```
TCompose_tag := rec(
        forTransposition := false,
        applicable := (self, nt) >> true,
        children := nt ->
            [ List(nt.params[1], e -> e.withTags(nt.getTags())) ],
        apply := (nt, c, nt) -> Grp(Compose(c))
)
```

TTensorI: Lift Tensor to Transform Level

Definition

```
# spiral-core\namespaces\spiral\paradigms\common\nonterminal.gi
Class(TTensorI, Tagged_tSPL_Container, rec(
    abbrevs :=  [ (nt, s, l, r) -> Checked(IsSPL(nt), IsPosInt(s),
        l in [APar, AVec], r in [APar, AVec], [nt, s, l, r]) ],
    dims := self >> self.params[1].dims()*self.params[2],
    terminate := self >> let(P := self.params, A:=P[1].terminate(),
        n:= P[2], l:=P[3], r:=P[4], Cond(
            IsParPar(P), Tensor(I(n), A),
            IsVecVec(P), Tensor(A, I(n)),
            IsParVec(P), Tensor(I(n), A) * L(A.dims()[2]*n, n),
            IsVecPar(P), let(m:=A.dims()[2],
                Tensor(A, I(n)) * L(m*n,m)))),
    transpose := self >> let(p := self.params,TensorI(p[1].transpose(),
        p[2], p[4], p[3]).withTags(self.getTags())),
    ...
));
```

TTensorI -> Tensor

```
TTensorI(F(2), 4, AVec, AVec).terminate();
TTensorI(F(2), 4, APar, APar).terminate();
TTensorI(F(2), 4, AVec, APar).terminate();
TTensorI(F(2), 4, APar, AVec).terminate();
```

Tagged Algorithmic Breakdown Rules

tSPL DFT_CT Breakdown Rule

```
# spiral-core\namespaces\spiral\paradigms\common\dft.gi
DFT_tSPL_CT := rec(
    maxSize  := false,
    filter := e->true,
    applicable := (self, nt) >> nt.params[1] > 2
        and (self.maxSize = false or nt.params[1] <= self.maxSize)
        and not IsPrime(nt.params[1])
        and nt.hasTags(),
    children := (self, nt) >> Map2(Filtered(
        DivisorPairs(nt.params[1]), self.filter), (m, n) -> [
        TCompose([
            TGrp(TCompose([
                TTensorI(DFT(m, nt.params[2] mod m), n, AVec, AVec),
                TTwiddle(m*n, n, nt.params[2])
            ])),
            TGrp(TTensorI(DFT(n, nt.params[2] mod n), m, APar, AVec))
        ]).withTags(nt.getTags())
    ]),
    apply := (nt, c, cnt) -> c[1],
    switch := false
  )
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

  Generalized Tensors

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Generalized Tensors: GT

GT Non-Terminal

```
# spiral-core\namespaces\spiral\paradigms\common\gt.gi
Class(GT, GTBase, Tagged_tSPL, rec(
    abbrevs := [
    (spl, gath, scat, v) -> Checked(IsSPL(spl),
        IsIndexMapping(gath), IsIndexMapping(scat), IsList(v),
        ForAll(v, IsPosInt0Sym), [spl, gath, scat, v] ) ],
    ...
    _scat := Scat,
    _gath := Gath,
    toSpl := self >> self.toSplCx([]),
    toSplCx := (self, outer_inds) >> let(p := self.params,
        spl := Copy(p[1]), g := Copy(p[2]), s := Copy(p[3]),
        dims := p[4], inds := List(dims, Ind),
        allinds := Concatenation(inds, outer_inds),
        kernel := self._scat(s.toSpl(allinds, Rows(spl))) * spl *
            self._gath(g.toSpl(allinds, Cols(spl))),
        kerneld := Cond(inds=[], kernel,
            SubstTopDownNR(kernel, @.cond(
                e->IsFunction(e) or IsFuncExp(e)),
                e->e.downRankFull(allinds))),
        FoldL(inds, (ker, idx) ->
            ISum(idx.setAttr("GT"), ker), kerneld))
);
```

XChains: Specialized Symbolic Functions

XChain Definition

```
# spiral-core\namespaces\spiral\spl\gtfuncs.gi
Class(XChain, GTIndexFunction, rec(
    def := perm -> Checked(IsList(perm), ForAll(perm, IsPosInt0),
        Set(Copy(perm))=[0..Length(perm)-1], rec()),
    range := self >> 0,
    domain := self >> 0,
    equals := (self, o) >> ObjId(self)=ObjId(o) and
        self.params[1]=o.params[1],
    toSpl := (self, inds, kernel_size) >> let(fbases := List(inds,
        fBase), ApplyFunc(fTensor, List(self.params[1],
        i -> When(i=0, fId(kernel_size), fbases[i])))),
    ...
));
```

Convert GT and XChain to SPL and Symbolic Functions

```
x := XChain([0,2,1]);
x.toSpl([Ind(8), Ind(4)], 4);
gt1 := GT(DFT(2), XChain([ 0, 1 ]), XChain([ 0, 1 ]), [ 4 ]);
gt2 := GT(DFT(2), XChain([ 1, 0 ]), XChain([ 1, 0 ]), [ 4 ]);
gt3 := GT(DFT(2), XChain([ 0, 1 ]), XChain([ 1, 0 ]), [ 4 ]);
gt4 := GT(DFT(2), XChain([ 1, 0 ]), XChain([ 0, 1 ]), [ 4 ]);
gt5 := GT(DFT(2), XChain([ 2, 1, 0 ]), XChain([ 0, 2, 1 ]), [ 2, 4 ]);
```

GT Breakdown Rules

Convert TTensorI to GT

```
TTensorI_toGT := rec(
    applicable := t -> true,
    freedoms := t -> [],
    child := (t, fr) -> [ GT_TTensorI(t) ],
    apply := (t, C, Nonterms) -> C[1]
);
```

Helper Functions

```
# spiral-core\namespaces\spiral\paradigms\common\gt.gi
GTVec := XChain([0,1]);
GTPar := XChain([1,0]);

GT_TTensorI := function(tt)
    local spl, g, s, v, tags;
    [spl,v,s,g] := tt.params;
    tags := tt.getTags();
    g := When(g=AVec, GTVec, GTPar);
    s := When(s=AVec, GTVec, GTPar);
    return GT(spl, g, s, [v]).withTags(tags);
end;
```

GT Parallelization Rule

Very dense—handles many cases and options

```
# spiral-core\namespaces\spiral\paradigms\common\breakdown.gi
GT_Par := rec(requiredFirstTag:=AParSMP,
applicable := (self, t) >> let(rank := Length(t.params[4]),
    nthreads:=t.firstTag().params[1],rank=1 and let(its:=t.params[4][1],
    PatternMatch(t, [GT,@(1),@(2,XChain),@(3,XChain),...],empty_cx())
    and IsPosInt(its/nthreads))),
children := (self, t) >> let(spl := t.params[1], g := t.params[2],
    s := t.params[3], its := t.params[4][1],
    nthreads := t.firstTag().params[1], tags := Drop(t.getTags(), 1),
    [[ GT(spl,g,s,[its / nthreads]).withTags(tags), InfoNt(nthreads) ],
     [ GT(spl,XChain([0]),XChain([0]),[]).withTags(tags),InfoNt(its) ]],
apply := (self, t, C, Nonterms) >> let(
    spl := t.params[1], N := Minimum(spl.dimensions),
    g := t.params[2], s := t.params[3], its := t.params[4][1],
    gg := When(g.params[1]=[0,1], XChain([0,1,2]), XChain([1,2,0])),
    ss := When(s.params[1]=[0,1], XChain([0,1,2]), XChain([1,2,0])),
    nthreads := t.firstTag().params[1],tid:=t.firstTag().params[2],
    par_its:=When(IsBound(Nonterms[2]),Nonterms[2].params[1],nthreads),
    i := Ind(par_its), SMPBarrier(nthreads, tid,
    SMPSum(nthreads, tid, i, par_its,
        Scat(ss.part(1, i, Rows(spl), [par_its, its/par_its])) *
        C[1]*Gath(gg.part(1, i, Cols(spl), [par_its, its/par_its])))))
)
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**
  Tagged SPL and Σ-SPL objects

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

SMP Tagged SPL/Σ-SPL Objects

Parallel Loop == SMPSum

```
# spiral-core\namespaces\spiral\paradigms\smp\sigmaspl.gi
Class(SMPSum, ISum, rec(
    doNotMarkBB := true,
    abbrevs := [ (p, var, domain, spl) -> Checked(IsInt(p) or
        IsScalar(p), IsVar(var), IsInt(p) or IsScalar(domain),
        IsSPL(spl), [p, threadId(), var, domain, spl]) ],
    new := meth(self, nthreads, tid, var, domain, spl)
        local res;
        Constraint(IsSPL(spl));
        Constraint(IsPosIntSym(domain));
        var.isLoopIndex := true;
        var.range := domain;
        res := SPL(WithBases(self, rec(nthreads:=nthreads, tid:=tid,
            _children := [spl], var := var, domain := domain)));
        res.dimensions := res.dims();
        return res;
    end,
    print := (self, i, is) >> Print(self.__name__, "(",
        self.nthreads, ", ", self.tid, ", ", self.var, ", ",
        self.domain, ",\n",
        Blanks(i+is), self.child(1).print(i+is, is), "\n",
        Blanks(i), ")", self.printA())
));
```

SMP Tagged SPL/Σ-SPL Objects

Barrier Object

```
# spiral-core\namespaces\spiral\paradigms\smp\sigmaspl.gi
Class(SMPBarrier, Buf, BaseContainer, rec(
    doNotMarkBB := true,
    new := (self, nthreads, tid, spl) >> Checked(IsPosInt0Sym(tid),
        IsPosIntSym(nthreads), IsSPL(spl),
        SPL(WithBases(self, rec(_children:=[spl],
        tid := tid, nthreads := nthreads, dimensions := spl.dims())))),
    dims := self >> self._children[1].dims(),
));
```

Context Object

```
Class(SMP, BaseContainer, rec(
    doNotMarkBB := true,
    abbrevs := [ (nthreads, spl) -> Checked(IsInt(nthreads) or
        IsScalar(nthreads), IsSPL(spl), [nthreads, threadId(), spl]) ],
    new := (self, nthreads, tid, spl) >> SPL(WithBases(self,
        rec(nthreads:=nthreads, tid:=tid,
        dimensions := spl.dimensions, _children := [spl]))),
    sums := self >> ObjId(self)(
        self.nthreads, self.tid, self.child(1).sums()),
));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

  Architecture specific icode and unparsing

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

SMP Tagged Code Objects

Parallel Loop == smp_for

```
# spiral-core\namespaces\spiral\paradigms\smp\code.gi
Class(smp_loop, loop_base, rec(
    __call__ := meth(self, nthreads, tidvar, tidexp,
                     loopvar, range, cmd)
        local result;
        range := toRange(range);
        loopvar.setRange(range);
        loopvar.isLoopIndex := true;
        return WithBases(self, rec(
            operations := CmdOps,
            nthreads := nthreads,
            cmd := cmd,
            var := loopvar,
            tidvar := Checked(IsLoc(tidvar), tidvar),
            tidexp := toExpArg(tidexp),
            range := range));
    end,
    print := (self, i, is) >> Print(self.name,"(",self.nthreads,", ",
        self.tidvar, ", ", self.tidexp, ", ", self.var, ", ",
        self.range, ",\n", Blanks(i+is),
        self.cmd.print(i+is, is),
        Print("\n", Blanks(i), ")")),
));
```

SMP Code Objects and Code Generator

Thread ID

```
# spiral-core\namespaces\spiral\paradigms\smp\sigmaspl.gi
Class(threadId, Exp, rec(computeType := self >> TInt));
```

Barrier

```
Class(barrier, call, rec(visitAs := call));
```

SMP Codegenerator

```
Class(SMPCodegenMixin, Codegen, rec(
    SMPBarrier := (self, o, y, x, opts) >> chain(
        self(o.child(1), y, x, opts),
        barrier(o.nthreads, o.tid, "&GLOBAL_BARRIER")),
    SMPSum := (self, o, y, x, opts) >> let(
        outer_tid      := When(IsBound(opts._tid), opts._tid, 0),
        outer_num_thr := When(IsBound(opts._tid), opts._tid.range, 1),
        tid := var.fresh("tid", TInt, o.nthreads * outer_num_thr),
        smp_loop(o.nthreads, tid, (outer_tid * outer_num_thr) + o.tid,
            o.var, o.domain,
            self(o.child(1), y, x,
                CopyFields(opts, rec(_tid := tid))))
        )
));
```

OpenMP Unparser

Unparser Definition

```
# Unparser for #pragma omp parallel for
Class(OpenMP_Unparser, OpenMP_UnparseMixin_ParFor, CUnparserProg);
```

Unparser Parallel For Mixin

```
# spiral-core\namespaces\spiral\paradigms\smp\unparsed.gi
Class(OpenMP_UnparseMixin_ParFor, SMP_UnparseMixin, rec(
    includes := ["<omp.h>"],
    threadId := (self,o,i,is) >> Print("omp_get_thread_num()"),
    barrier := (self,o,i, is) >>
        Print(Blanks(i), "/* SMP barrier */\n"),
    smp_loop := (self,o,i,is) >> let(v := o.var,
        lo := 0, hi := o.range,
        Print(Blanks(i),
        "#pragma omp parallel for schedule(static, ",
        Int((hi+1)/2),")\n",
        Blanks(i), "for(int ", v, " = ", lo, "; ", v,
        " < ", hi, "; ", v, "++) {\n",
        Blanks(i + is), "int ", o.tidvar, " = ", v, ";\n",
        self.opts.unparser(o.cmd,i+is,is),
        Blanks(i), "}\n")),
));
```

OpenMP Unparser

Unparser Definition

```
# Unparser for #pragma omp parallel regions
# spiral-core\namespaces\spiral\libgen\recgt.gi
Class(OpenMP_Unparser, OpenMP_UnparseMixin, CUnparserProg);
```

Unparser Parallel Region Mixin

```
# spiral-core\namespaces\spiral\paradigms\smp\unparsed.gi
Class(OpenMP_UnparseMixin, SMP_UnparseMixin, rec(
    includes := ["<omp.h>"],
    smp_fork := (self, o, i, is) >> Print(
        Blanks(i), "#pragma omp parallel num_threads(",
        o.nthreads, ")\n",
        Blanks(i), "{\n",
        self(o.cmd, i+is, is),
        Blanks(i), "}\n"
    ),
    threadId := (self,o,i,is) >> Print("omp_get_thread_num()"),
    barrier := (self,o,i, is) >> Print("#pragma omp barrier\n")
));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**
  Top level interface

- **Benchmark/production code infrastructure**

Targeting SIMD Vector Instructions

Simple Example: Intel AVX 4-way double precision

```
opts := SIMDGlobals.getOpts(AVX_4x64f); # default: real vectorization
t := TRC(DFT(16)).withTags(opts.tags);
rt := RandomRuleTree(t, opts);
c := CodeRuleTree(rt, opts);
PrintCode("AVX_DFT16", c, opts);
```

Stepwise code generation

```
opts.tags;                              # what are the tags
opts.tags[1].v;                         # vector length
opts.tags[1].isa;                       # targeted ISA
opts.vector;                            # check out the options used
spl := SPLRuleTree(rt);                 # There are SIMD SPL objects
s := SumsRuleTree(rt, opts);            # and a SMP ISum
InfinityNormMat(MatSPL(s) - MatSPL(t));     # correctness check
```

Complex Vectorization Example

```
optsc := SIMDGlobals.getOpts(AVX_4x64f, # __m256d = (re,im,re,im)
    rec(realVect := false, cplxVect := true));
rtc := RandomRuleTree(t, optsc);        # complex vectorized DFT(16)
sc := SumsRuleTree(rtc, optsc);
cc := CodeRuleTree(rtc, optsc);         # far fewer shuffle operations
PrintCode("AVXcplx_DFT16", cc, optsc);
```

Tags
Real Vectorization

```
# spiral-core\namespaces\spiral\paradigms\vector\tags.gi
Class(AVecReg, AGenericTag, rec(
    isReg := true, isRegCx := false, isVec := true,
    updateParams := meth(self)
        Checked(IsSIMD_ISA(self.params[1]));
        Checked(Length(self.params)=1);
        self.v := self.params[1].v; self.isa := self.params[1];
    end,
    container := (self, spl) >>
        paradigms.vector.sigmaspl.VContainer(spl, self.isa)
));
```

Complex Vectorization

```
Class(AVecRegCx, AVecReg, rec(
    updateParams := meth(self)
        Checked(IsSIMD_ISA(self.params[1]));
        Checked(Length(self.params)=1);
        self.v := self.params[1].v/2; self.isa := self.params[1];
    end,
    container := (self, spl) >>
        paradigms.vector.sigmaspl.VContainer(spl, self.isa.cplx()),
    isRegCx := true
));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

  Vectorization Rules

- **Benchmark/production code infrastructure**

Vectorization Rules: Simple Vectorization

Example Vectorization Rule

```
# spiral-core\namespaces\spiral\paradigms\vector\breakdown.gi
NewRulesFor(TTensorI, rec(
AxI_vec := rec(
    forTransposition := false,
    applicable := nt -> nt.hasTags() and IsVecVec(nt.params) and
        (nt.isTag(1,AVecReg) or nt.isTag(1,AVecRegCx)) and
        IsInt(nt.params[2]/nt.firstTag().v),
    children := nt -> let(r := nt.params[2] / nt.firstTag().v,
            isa := nt.firstTag().isa,
            [[ When(r = 1,
                When(nt.numTags() = 1,
                    nt.params[1].setWrap(VWrap(isa)),
                    nt.params[1].setWrap(
                        Drop(nt.getTags(), 1)).setWrap(VWrap(isa))
                ),
                TTensorI(nt.params[1].setWrap(VWrap(isa)), r,
                    AVec, AVec).withTags(Drop(nt.getTags(), 1))
            )]]
        ),
    apply := (nt, c, cnt) -> VTensor(c[1], nt.firstTag().v)
  )
));
```

Vectorization Rules: Formula Rewrite

Kronecker Commute

```
# spiral-core\namespaces\spiral\paradigms\vector\breakdown.gi
NewRulesFor(TTensorI, rec(
    IxA_vec := rec(forTransposition := false,
    applicable := nt -> IsParPar(nt.params) and nt.hasTags() and
        (nt.isTag(1,AVecReg) or nt.isTag(1,AVecRegCx)) and
        IsInt(nt.params[2]/nt.firstTag().v),
    children := nt -> let(pv := nt.getTags(), v := pv[1].v,
        isa := pv[1].isa, d := nt.params[1].dims(),
        [[
            TL(d[1]*v, v, 1, 1).withTags(pv).setWrap(VWrapId),
            When(Length(pv)=1, nt.params[1].setWrap(VWrap(isa)),
                nt.params[1].setpv(Drop(pv, 1)).setWrap(VWrap(isa))),
            TL(d[2]*v, d[2], 1, 1).withTags(pv).setWrap(VWrapId)
        ]]),
    apply := (nt, c, cnt) -> let(
        l := nt.params[2] / nt.firstTag().v,
        A := c[1] * VTensor(c[2], nt.firstTag().v) * c[3],
        NoDiagPullin(When(l=1, A , Tensor(I(l), A)))))
    )
));
```

Wrapping

Needed in Context of Search

```
# spiral-core\namespaces\spiral\paradigms\vector\vwrap.gi
Class(VWrap, VWrapBase, rec(
    __call__ := (self,isa) >> Checked(IsSIMD_ISA(isa),
        WithBases(self, rec(operations:=PrintOps, isa:=isa))),
    wrap := (self,r,t,opts) >> let(
        isa := self.isa, v := isa.v,
        tt := When(t.isReal(),
            @_Base(paradigms.vector.sigmaspl.VTensor(r.node, v), r),
                paradigms.vector.breakdown.AxI_vec(
                    TTensorI(TRC(t).withTags([AVecReg(isa)]), v,
                        AVec, AVec).withTags([AVecReg(isa)]),
                paradigms.vector.breakdown.TRC_VRCLR(
                    TRC(t).withTags([AVecReg(isa)]), r))),
    print := self >> Print(self.name, "(", self.isa, ")"),
));
```

VWrap Transformation, used in DP to time sub-trees

```
opts := SIMDGlobals.getOpts(AVX_4x64f);
t := DFT(2).setWrap(VWrap(AVX_4x64f));
rt := RandomRuleTree(t, opts);
wrt := t.wrap.wrap(rt, t, opts);
SPLRuleTree(wrt);
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

  Vectorization of Stride Permutation

- **Benchmark/production code infrastructure**

Special Role of Stride Permutations

TL: Lift Stride Permutation to Non-Terminal Level

```
# spiral-core\namespaces\spiral\paradigms\common\nonterms.gi
Class(TL, Tagged_tSPL_Container, rec(
    abbrevs :=  [ (size, stride) -> Checked(ForAll([size, stride],
        IsPosIntSym), [size, stride, 1, 1]),
            (size, stride, left, right) ->
        Checked(ForAll([size, stride, left, right], IsPosIntSym),
            [size, stride, left, right]) ],
    __call__  := arg >> let(self := arg[1], args := Drop(arg, 1),
        Cond(args=[1,1,1,1], I(1), ApplyFunc(Inherited,args))),
    dims := self >>
        Replicate(2, self.params[1]*self.params[3]*self.params[4]),
    terminate := self >> Tensor(I(self.params[3]),
        L(self.params[1], self.params[2]), I(self.params[4])),
    transpose := self >> TL(self.params[1],
        self.params[1]/self.params[2], self.params[3],
        self.params[4]).withTags(self.getTags()),
));
```

VWrap Transformation, used in DP to time sub-trees

```
t := TL(8,2,2,4);
t.terminate();
```

Architecture Specific Permutations

Looking up vectorized Implementations for TL

```
Import(paradigms.vector.sigmaspl);
opts := SIMDGlobals.getOpts(AVX_4x64f);
opts.breakdownRules.TL;
t := TL(16,4,1,1).withTags(opts.tags);        # a in-register perm
rt := RandomRuleTree(t, opts);
HashLookup(opts.baseHashes[1], t);     # the implementation is cached

s := SPLRuleTree(rt);
vp := Collect(s, VPerm)[1];                # SPL object carries its code
vp.code;                                    # code generator refers to ISA
AVX_4x64f.rules;                            # ISA carries implementations
PrintCode("", vp.code(Y, X), opts);    # for in-register perms
```

SIMD ISA Database and bootstrapping a vector architecture

```
SIMD_ISA_DB;                          # central SIMD data base
SIMD_ISA_DB.installed();      # all the ISAs supported
Doc(AVX_4x64f);                       # The ISA carries all the relevant info
Print(SIMD_ISA_DB.buildBases);       # How the base cases are built
AVX_4x64f.buildRules;         # bootstrapping function
SIMD_ISA_DB.getBases(AVX_4x64f);     # all the base cases needed
SIMD_ISA_DB.lookupBases(AVX_4x64f);  # and how they are implemented
```

ISA Database and Hashed Breakdowns

Generic Breakdown Rules to look up architecture specific code

```
# spiral-core\namespaces\spiral\paradigms\vector\bases\tl_bases.gi
NewRulesFor(TL, rec(
    SIMD_ISA_Bases1 := rec(
        forTransposition := false,
        applicable := (self, t) >> t.isTag(1, AVecReg) and
            let(isa := t.firstTag().isa, P:=t.params,
                isa.active and ForAny(isa.supportedTL(),
                e -> _TL_applicable(e, P[1], P[2], P[3], P[4]))),
        apply := function(nt,C,cnt)
            local isa, tl, ll, vprm, P;
            P:=nt.params;
            isa := nt.firstTag().isa;
            tl := isa.getTL(P);
            ll := P[3] / tl.perm.l;
            vprm := tl.vperm;
            return When(ll = 1, vprm,
                BlockVPerm(ll, isa.v, vprm, tl.perm.spl));
        end,
    )
));
```

Stride Permutatiopn Identities as Rules

Generic Breakdown Rules to look up architecture specific code

```
# spiral-core\namespaces\spiral\paradigms\common\breakdown.gi
NewRulesFor(TL, rec(
    IxLxI_kmn_n := rec (forTransposition := false,
        applicable := nt ->
            Length(DivisorsIntDrop(nt.params[1]/nt.params[2])) > 0,
        children := nt -> let(
            N := nt.params[1], n := nt.params[2],
            km := N/n, ml := DivisorsIntDrop(km),
            l := nt.params[3], r := nt.params[4],
            List(ml, m -> let( k := km/m, [
                    TL(k*n, n, l, r*m).withTags(nt.getTags()),
                    TL(m*n, n, k*l, r).withTags(nt.getTags())
                ]))
            ),
        apply := (nt, c, cnt) -> let(
            spl := c[1] * c[2],
            When(nt.params[1] = nt.params[2]^2,
                SymSPL(spl),
                spl
            )
        )
    )
));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

  Vector SPL/Σ-SPL

- **Benchmark/production code infrastructure**

SPL And Σ-SPL Vector Objects

Generate The Example

```
Import(paradigms.vector.sigmaspl);
opts := SIMDGlobals.getOpts(AVX_4x64f);
rt := RandomRuleTree(TRC(DFT(16)).withTags(opts.tags), opts);
s := SPLRuleTree(rt);                  # SPL Objects
ss := SumsRuleTree(rt, opts);          # Sigma-SPL Objects
```

Inspecting the Vector Objects

```
Collect(s, VTensor)[1];        # Vectorized Tensor(., I(v))
Collect(ss, VTensor)[1];       # Vectorized Tensor(., I(v))
Collect(s, VPerm)[1];          # Vectorized Prm(.)
Collect(s, BlockVPerm)[1];     # Vectorized Tensor(I(.), Prm(.))
Collect(s, VContainer)[1];     # Provides context for rewriting
Collect(s, VRC)[1];            # Carries interleaved complex format
Collect(ss, VGath)[1];         # Tensor(Gath(.), I(v))
Collect(ss, VScat)[1];         # Tensor(Scat(.), I(v))
Collect(ss, VRCDiag)[1];       # Vectorized Diag(.)
```

Vector SPL Objects

Vector Tensor SPL Object

```
# spiral-core\namespaces\spiral\paradigms\vector\sigmaspl\vtensor.gi
Class(VTensor, Tensor, rec(
    new := (self, L) >> SPL(WithBases(self, rec(
        _children := [L[1]],
        dimensions := When(IsBound(L[1].dims), L[1].dims(),
        L[1].dimensions) * L[2], vlen := L[2]))),
    from_rChildren := (self, rch) >> ObjId(self)(rch[1], self.vlen),
    print := (self,i,is) >> Print(self.name, "(",
        self.child(1).print(i+is,is), ", ", self.vlen, ")"),
    toAMat := self >> Tensor(self.child(1), I(self.vlen)).toAMat(),
    sums := self >> Inherit(self, rec(_children :=
        [self.child(1).sums()])),
    isPermutation := False,
    dims := self >> self.child(1).dims() * self.vlen,
    needInterleavedLeft := False,
    needInterleavedRight := False,
    transpose := self >> VTensor(self.child(1).transpose(), self.vlen),
    isBlockTransitive := true,
    cannotChangeDataFormat := False,
));
```

Spiral
Software/Hardware Generation for Performance

SpiralGen
Accelerating Innovation in Computing

Vector Σ-SPL Objects
Vector Gather

```
# spiral-core\namespaces\spiral\paradigms\vector\sigmaspl\gather.gi
Class(VGath, BaseVGath, SumsBase, rec(
    rChildren := self >> [self.func],
    rSetChild := rSetChildFields("func"),
    from_rChildren := (self, rch) >> ObjId(self)(rch[1], self.v),
    new := (self, func, v) >> SPL(WithBases(self,
        rec(func := func, v := v))).setDims(),
    dims := self >> [self.v*self.func.domain(),
        self.v*self.func.range()],
    transpose := self >> VScat(self.func, self.v),
    print := (self,i,is) >> Print(self.name, "(", self.func, ", ",
        self.v,")", self.printA()),
    toAMat := self >> let(v:=self.v, n :=
        EvalScalar(v*self.func.domain()),
            N := EvalScalar(v*self.func.range()),
            func := fTensor(self.func, fId(v)).lambda(),
            AMatMat(List([0..n-1], row -> BasisVec(N,
                EvalScalar(func.at(row).ev())))))),
));
```

Vector RC Objects: Manage Data Layout

Vector RC

```
# spiral-core\namespaces\spiral\paradigms\vector\sigmaspl\vrc.gi
Class(VRC, RC, rec(
    toAMat := (self) >> AMatMat(RCMatCyc(MatSPL(self.child(1)))),
    new := meth(self, spl, v)
        local res;
        res := SPL(WithBases(self, rec(_children:=[spl], v:=v,
                                    dimensions := spl.dimensions)));
        res.dimensions := res.dims();
        return res;
    end,
    print := (self, i, is) >> Print(self.__name__,
        "(\n", Blanks(i+is), self.child(1).print(i+is,is), ", ",
        #"\n", Blanks(i+is),
        self.v,
        #"\n", Blanks(i),
        ")", self.printA()),
    unroll := self >> self,
    transpose := self >> VRC(self.child(1).conjTranspose(), self.v),
    vcost := self >> self.child(1).vcost(),
    from_rChildren := (self, rch) >> ObjId(self)(rch[1], self.v)
));
```

Vector RC Objects: Manage Data Layout

Vector RC

```
v1 := VRC(Tensor(I(2), F(2)), 4);        # Implicit data reorganization
MatSPL(v1);
v2 := VRCL(Tensor(I(2), F(2)), 4);       # The L and R encodes
MatSPL(v2);
v3 := VRCR(Tensor(I(2), F(2)), 4);       # which side goes from
MatSPL(v3);
v4 := VRCLR(Tensor(I(2), F(2)), 4);      # interleaved complex format to
MatSPL(v4);                              # block split complex format
```

Terminate VRC, VRCL, VRCR, VRCLR

```
Import(paradigms.vector.rewrite);
opts := SIMDGlobals.getOpts(AVX_4x64f);
# see how the format gets propagated down the tree
v5 := VRC(Tensor(I(2), F(2)) * Tensor(F(2), I(2)), 4);
RulesVRC(v5);
# when propagated to the leftmost/rightmost tree leaves, terminate
v6 := VRCL(VTensor(F(2), 4), 4);
v7 := Rewrite(v6, [RulesVRCTerm], opts);
# termnination inserts VPerms to implement local data format change
v8 := VRCR(VTensor(F(2), 4), 4);
v9 := Rewrite(v9, [RulesVRCTerm], opts);
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**
  SumsGen, Codegen and Unparser

- **Benchmark/production code infrastructure**

Vector SumsGen and Rewriting

Default Sumsgen Handles Vector SPL to Σ-SPL

```
opts := SIMDGlobals.getOpts(AVX_4x64f);
opts.sumsgen;
opts.sumsgen.VTensor;
opts.sumsgen.VPerm;
```

Rewrite Rule Strategies

```
opts.formulaStrategies;
opts.formulaStrategies.sigmaSpl;
opts.formulaStrategies.preRC;
opts.formulaStrategies.postProcess;
```

Compile Strategies

```
opts.compileStrategy;
Print(opts.vector.isa.fixProblems);
```

Vector CodeGen
Default Sumsgen Handles SPL to Σ-SPL

```
# spiral-core\namespaces\spiral\paradigms\vector\sigmaspl\codegen.gi
Class(VectorCodegen, DefaultCodegen, rec(
    VContainer := (self, o, y, x, opts) >>
        self(o.child(1), y, x, CopyFields(opts, rec(
            vector := rec(
                isa  := o.isa,
                SIMD := LocalConfig.cpuinfo.SIMDname )))),
    VPrm_x_I := (self, o, y, x, opts) >>
        self(VTensor(Prm(o.func), o.v), y, x, opts),
    VPerm := (self, o, y, x, opts) >> o.code(y, x),
    VTensor := (self, o, y, x, opts) >> let(
        CastToVect := p -> StripList(List(Flat([p]), e ->
        tcast(TPtr(TVect(opts.vector.isa.t.t, o.vlen)), e))),
        self(o.child(1), CastToVect(y), CastToVect(x), opts)),
    VGath := (self, o, y, x, opts) >> Cond(IsUnalignedPtrT(x.t),
        self(VGath_u(fTensor(o.func, fBase(o.v, 0)), o.v), y, x, opts),
        self(VTensor(Gath(o.func), o.v), y, x, opts)),
    VScat := (self, o, y, x, opts) >> Cond(IsUnalignedPtrT(y.t),
        self(VScat_u(fTensor(o.func, fBase(o.v, 0)), o.v), y, x, opts),
        self(VTensor(Scat(o.func), o.v), y, x, opts))
));
```

Vector Instructions
Every ISA defines ISA specific instructions and polymorphic add,…

```
# spiral-core\namespaces\spiral\platforms\avx\code.gi
# __m256d _mm256_insertf128_pd(__m256d a, __m128d b, int offset);
Class(vinsert_2l_4x64f, VecExp_4.binary(), rec(
    ev := self >> let(
        a := _unwrap(self.args[1].ev()),
        b := _unwrap(self.args[2].ev()),
        When( self.args[3].p[1] = 0,
            b :: a{[3 .. 4]}, a{[1 .. 2]} :: b )),
    computeType := self >> self.args[1].t,
));

# __m256d _mm256_unpackhi_pd(__m256d a, __m256d b);
Class(vunpackhi_4x64f, VecExp_4.binary(), rec(
    semantic := (in1, in2, p) -> [in1[2], in2[2], in1[4], in2[4]],
    ev := _evpack
));

#__m256 _mm256_blend_ps(__m256 m1, __m256 m2, const int mask);
Class(vblend_8x32f,  VecExp_8.binary(), rec(
    semantic := (in1, in2, p) ->
    List( Zip2(TransposedMat([in1, in2]), p), e -> e[1][e[2]]),
    params := self >> Replicate(8, [1,2]), ev := _evshuf2
));
```

Vector Strength Reduction
Strength Reduction and Fixup Rules

```
# spiral-core\namespaces\spiral\platforms\avx\sreduce.gi
RewriteRules(FixCodeAVX, rec(
    fix_noneExp := Rule( noneExp, e -> e.t.zero()),
    vpermf128_8x32f_to_vextract := Rule( [assign, [deref, @(1)],
        [vpermf128_8x32f, @(2), @(3), @(4)]],
        e -> let( p := @(4).val.p,
            a := [[@(2).val, [0]], [@(2).val, [1]], [@(3).val,
            [0]], [@(3).val, [1]]],
            dst := tcast(TPtr(TVect(T_Real(32), 4)), @(1).val),
            chain(
                assign(deref(dst),
                    ApplyFunc(vextract_4l_8x32f, a[p[1]])),
                assign( deref(dst+1),
                    ApplyFunc(vextract_4l_8x32f, a[p[2]])))
        )),
    addsub_4x64f_to_mul := Rule( [addsub_4x64f, _0, @(1)],
        e -> mul(e.t.value([-1,1,-1,1]), @(1).val)),
    addsub_8x32f_to_mul := Rule( [addsub_8x32f, _0, @(1)],
        e -> mul(e.t.value([-1,1,-1,1,-1,1,-1,1]), @(1).val)),
    avx_add_addsub_vzero := Rule([add, @(1), [@(2, [addsub_4x64f,
        addsub_8x32f]), _0, @(3)]],
        e -> ObjId(@(2).val)(@(1).val, @(3).val)),
));
```

Vector Unparser

Polymorphic Unparsing for standard icode, adds special instructions

```
# spiral-core\namespaces\spiral\platforms\avx\unparse.gi
Class(AVXUnparser, SSEUnparser, rec(
    TVect := (self, t, vars, i, is) >> let(
        ctype := self.ctype(t, _isa(self)),
        Print(ctype, " ", self.infix(vars, ", "))),
    vpack := (self, o, i, is) >> Cond( _avxT(o.t, self.opts),
            Print("_mm256_set_", self.ctype_suffix(o.t, _isa(self)),
        "(", self.infix(Reversed(o.args), ", "), ")"),
        Inherited(o, i, is)),
    sub := (self, o, i, is) >> Cond( _avxT(o.t, self.opts), let(
        isa := _isa(self),
        sfx := self.ctype_suffix(o.t, isa),
        saturated := When(isa.isFixedPoint and
            isa.saturatedArithmetic, "s", ""),
            self.printf("_mm256_sub$1_$2($3, $4)", [saturated, sfx,
                o.args[1], o.args[2]])),
        Inherited(o, i, is)),
    vextract_2l_4x64f := (self, o, i, is) >>
        self.prefix("_mm256_extractf128_pd", o.args),
    vstore_2l_4x64f := (self, o, i, is) >>
        self.prefix("_mm256_extractf128_pd", o.args),
));
```

ISA Definition

ISA Definition file ties everyting together

```
# spiral-core\namespaces\spiral\platforms\avx\isa.gi
Class(AVX_4x64f, AVX_Intel, rec(
    includes    := () -> ["<include/omega64.h>"] :: _AVXINTRIN(),
    v        := 4,
    t        := TVect(T_Real(64), 4),
    ctype    := "double",
    instr    := [ vunpacklo_4x64f, vunpackhi_4x64f, vshuffle_4x64f,
        vperm2_4x64f, vpermf128_4x64f, vperm_4x64f, vblend_4x64f ],
    mul_cx := (self, opts) >>
        ((y, x, c) -> let( u1 := self.freshU(), u2 := self.freshU(),
            u3 := self.freshU(),
            decl([u1, u2, u3], chain(
                assign(u1, mul(x, vunpacklo_4x64f(c, c))),
                assign(u2, vshuffle_4x64f(x, x, [2,1,2,1])),
                assign(u3, mul(u2, vunpackhi_4x64f(c, c))),
                assign(y, addsub_4x64f(u1, u3))))))),
    svload_init := (vt) -> [
        [ y,x,opts) -> let(u1 := var.fresh_t("U", TVect(vt.t, 2)),
            decl([u1], chain(
                assign(u1, vload1sd_2x64f(x[1].toPtr(vt.t))),
                assign(y, vinsert_2l_4x64f(vt.zero(), u1, [0]))))),
    ...
));
```

Organization

- **Installation**

- **GAP3 language**

- **Spiral objects and data types**

- **Spiral infrastructure**

- **Special hardware: SMP/OpenMP**

- **Special hardware: SIMD/AVX**

- **Benchmark/production code infrastructure**

Combining SIMD Vector and OpenMP

IAGlobals = SIMDGlobals + SMPGlobals

```
opts := LocalConfig.getOpts(
    rec(cpu := LocalConfig.cpuinfo,   # some of the params and defaults
        useSIMD := true,
        useSMP := true,
        dataType := T_Real(64),
        globalUnrolling := 128),
    rec(numproc := LocalConfig.cpuinfo.cores,
        api := "OpenMP"),
    rec(svct:=true,
        splitL:=false,
        oddSizes := false,
        stdTTensor := true,
        tsplPFA := false));
opts.vector;
opts.smp;
t := TRC(DFT(1000)).withTags(opts.tags);
rt := RandomRuleTree(t, opts);
c := CodeRuleTree(rt, opts);
PrintCode("DFT1000", c, opts);
```

DPBench Infrastructure

DP Benchmark Object

```
Doc(DPBench);
```

Using DPBench

```
opts := SIMDGlobals.getOpts(AVX_4x64f);
t := TRC(DFT(512)).withTags(opts.tags);
dpbench := DPBench.build([t], opts, rec(), "DFT512", rec());

dpbench.runRandomAll();        # Random ruletree through DPBench
dpbench.runAll();              # now run search

dpbench.generateCode(dpbench.transforms, "DFT512");
Exec("dir");
Exec("type DFT512_TRC_DFT512.c");
```

Vector Benchmarking Infrastructure

LocalConfig provides unit tests

```
LocalConfig.bench;
```

Create a test and run it

```
dpbench := LocalConfig.bench.AVX().4x64f.1d.dft_ic.medium();
dpbench.runAll();
```

Underlying infrastructure

```
# spiral-core\namespaces\spiral\platforms\avx\bench.gi
medium := _defaultSizes(s->doSimdDft(s, AVX_4x64f,
    rec(globalUnrolling := 128, tsplRader:=false,
    tsplBluestein:=false, tsplPFA:=false, oddSizes:=false,
    interleavedComplex := true, cplxVect := false, realVect := true)),
    List([4..16], i->2^i));


Print(doSimdDft);        # constructor from paradigms.vector
```

More Information:

www.spiral.net
www.spiralgen.com