Faust compiler presentation

Yann Orlarey, Stéphane Letz

1 avril 2021

The different steps

- Internal representations based as non-mutables trees and hash-consing
- Lex/Yacc parsing
- Program evaluation as a circuit of signals processors
- Symbolic propagation of signals through the circuit
- Signals normalisation and optimisation
- Interval calculus and signal typing
- Signals translation in imperative code (FIR)
- Code generation by the chosen backend

Trees

Non-mutables trees, hash-consing, DAG, mutable properties.

Trees properties: $t_1 = t_2 \Leftrightarrow M(t_1) = M(t_2)$

- symbols: tlib/symbol.hh+cpp
- nodes: tlib/node.hh+cpp
- trees: tlib/tree.hh+cpp, 2 types of recursivity (Bruijn + symbolic)
- constructors
- destructors
- properties

Trees: example of sequential composition A:B

```
gGlobal->BOXSEQ = symbol("BoxSeq");

Tree boxSeq(Tree x, Tree y)
{
    return tree(gGlobal->BOXSEQ, x, y);
}
bool isBoxSeq(Tree t, Tree& x, Tree& y)
{
    return isTree(t, gGlobal->BOXSEQ, x, y);
}
```

Trees: *de Bruijn* recursivity

```
Tree rec(Tree body)
{    return tree(gGlobal->DEBRUIJN, body); }

bool isRec(Tree t, Tree& body)
{   return isTree(t, gGlobal->DEBRUIJN, body); }

Tree ref(int level)
{   return tree(gGlobal->DEBRUIJNREF, tree(level)); }

bool isRef(Tree t, int& level)
{
    Tree u;
    if (isTree(t, gGlobal->DEBRUIJNREF, u)) {
        return isInt(u->node(), &level);
    } else {
        return false;
    }
}
```

Trees: *symbolic* recursivity

```
Tree rec(Tree var, Tree body) {
    Tree t = tree(gGlobal->SYMREC, var);
    t->setProperty(gGlobal->RECDEF, body);
    return t; }

bool isRec(Tree t, Tree& var, Tree& body) {
    if (isTree(t, gGlobal->SYMREC, var)) {
        body = t->getProperty(gGlobal->RECDEF);
        return true;
    } else {
        return false;
    }
}

Tree ref(Tree id) { return tree(gGlobal->SYMREC, id); }

bool isRef(Tree t, Tree& v) {
    return isTree(t, gGlobal->SYMREC, v); }
```

Lex/Yacc Parsing

- `parser/faustlexer.l`
- `parser/faustparser.y`
- `libcode.cpp/parseSourceFiles()`
- `parser/sourcereader.hh/SourceReader`
- environment
- recursive loader, using URLs

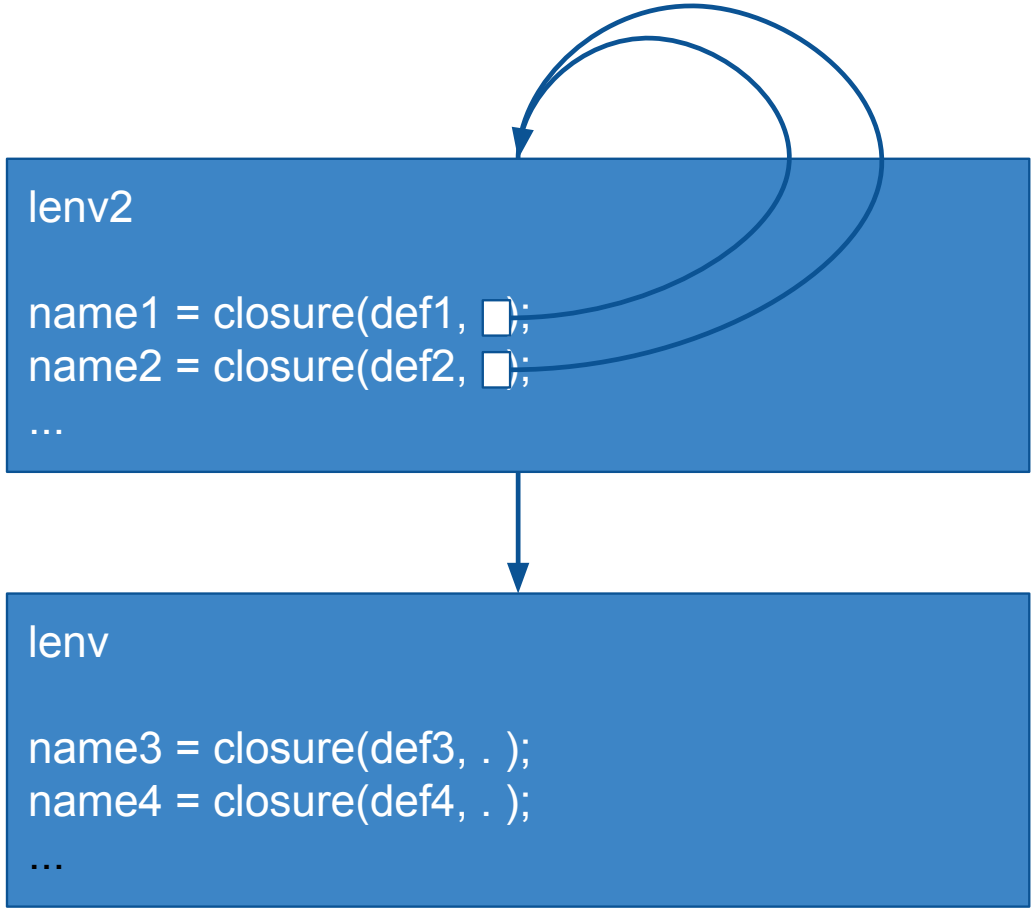
Parsing

Parsing of the `library("math.lib")` : expression

```
(G := gGlobal)
const char* fname = tree2str(label);
Tree eqlst = G->gReader.expandList(G->gReader.getList(fname));
Tree res   = closure(boxEnvironment(), G->nil, G->nil,
                     pushMultiClosureDefs(eqlst, G->nil, G->nil));
setDefNameProperty(res, label);
return res;
```

Environments

The definitions of a program are organized into environments by `pushMultiClosureDefs()` :



Evaluation

Evaluation of the definition of `process` in the environment resulting from reading the source files (see `eval.cpp`):

```
Tree evalprocess(Tree eqlist)
{
    Tree b=a2sb(eval(boxIdent(G->gProcessName.c_str()), G->nil,
                pushMultiClosureDefs(eqlist, G->nil, G->nil)));

    if (G->gSimplifyDiagrams) {
        b = boxSimplification(b);
    }

    return b;
}
```

Evaluation example

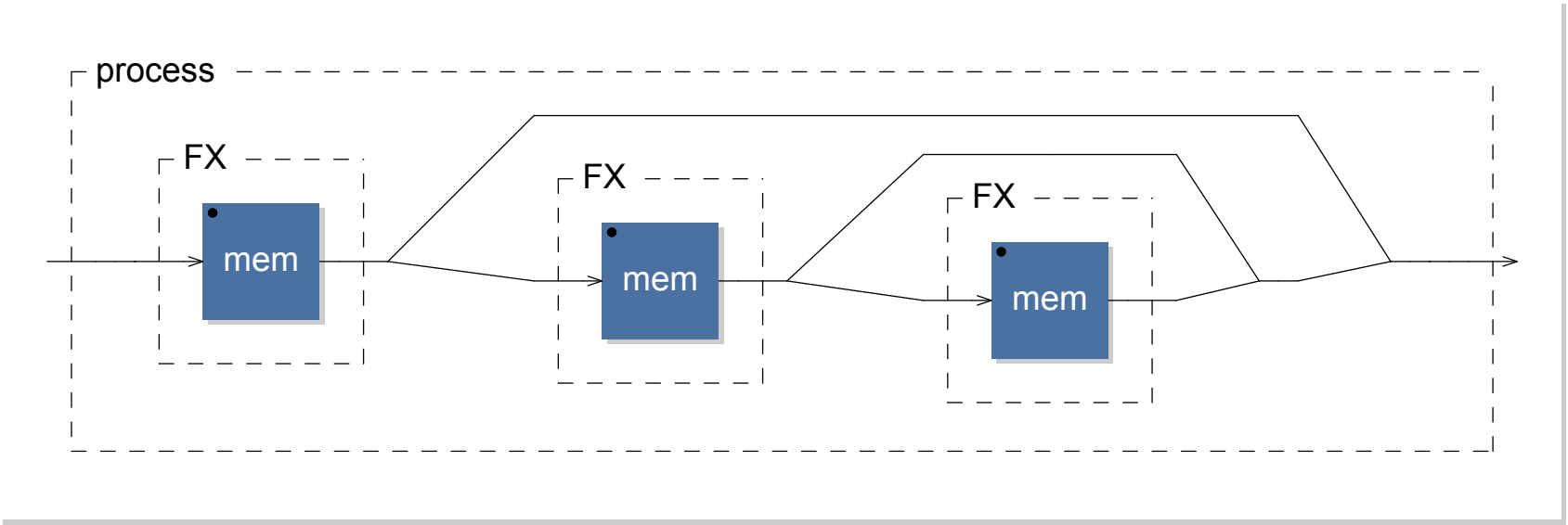
```
repeat(1,f) = f;
repeat(n,f) = f <: _, repeat(n-1,f) :> _;

N = 6/2;
FX = mem;
process = repeat(N,FX);
```

Normal Form

The result of the evaluation is a circuit *in normal form* where only a composition of primitives remains:

```
mem <: _, (mem <: _, mem :> _) :> _
```

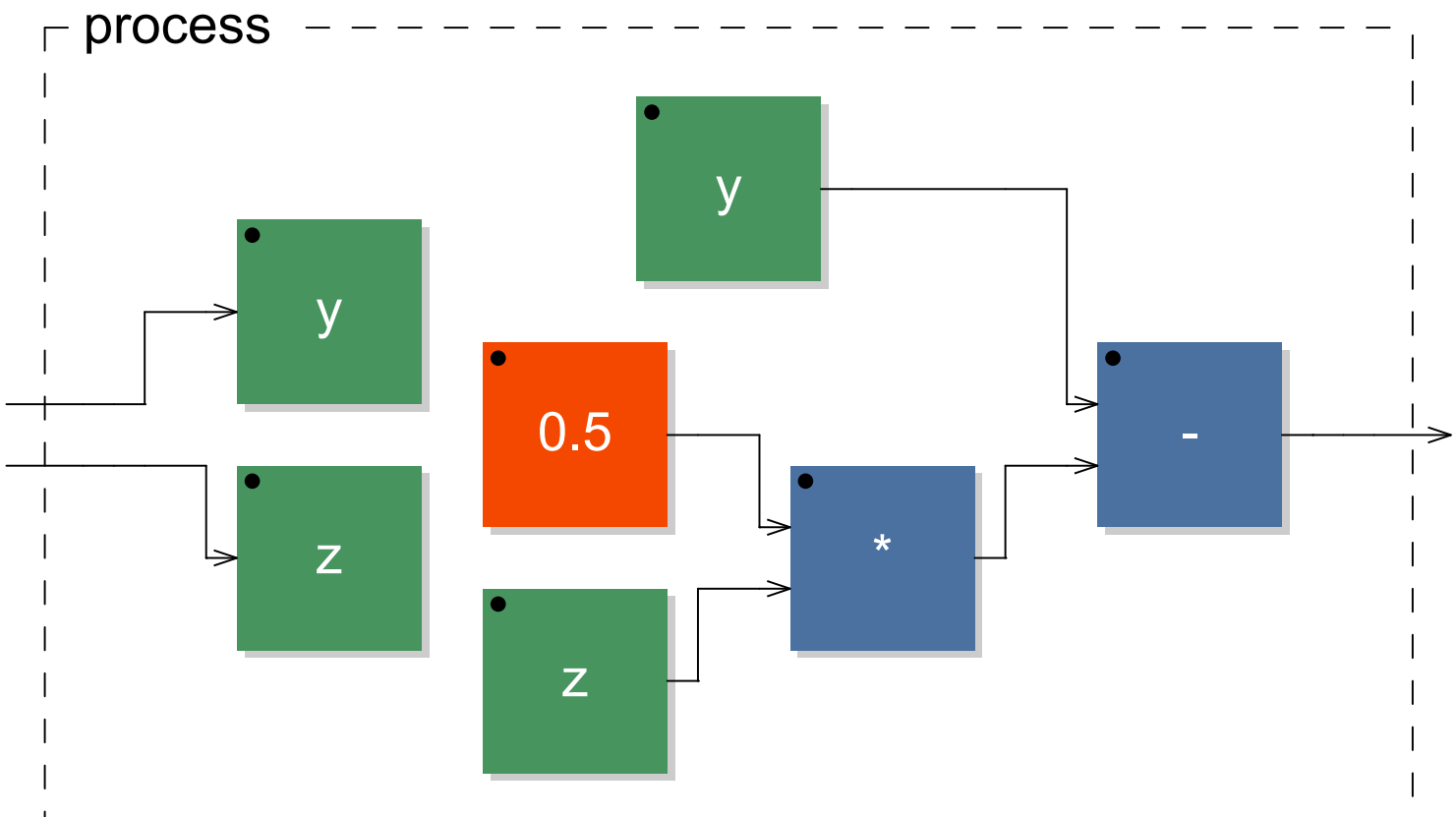
The SVG diagram is the graphic representation (possibly hierarchical) of the normal form:



The remaining abstractions (not applied) are transformed into routing

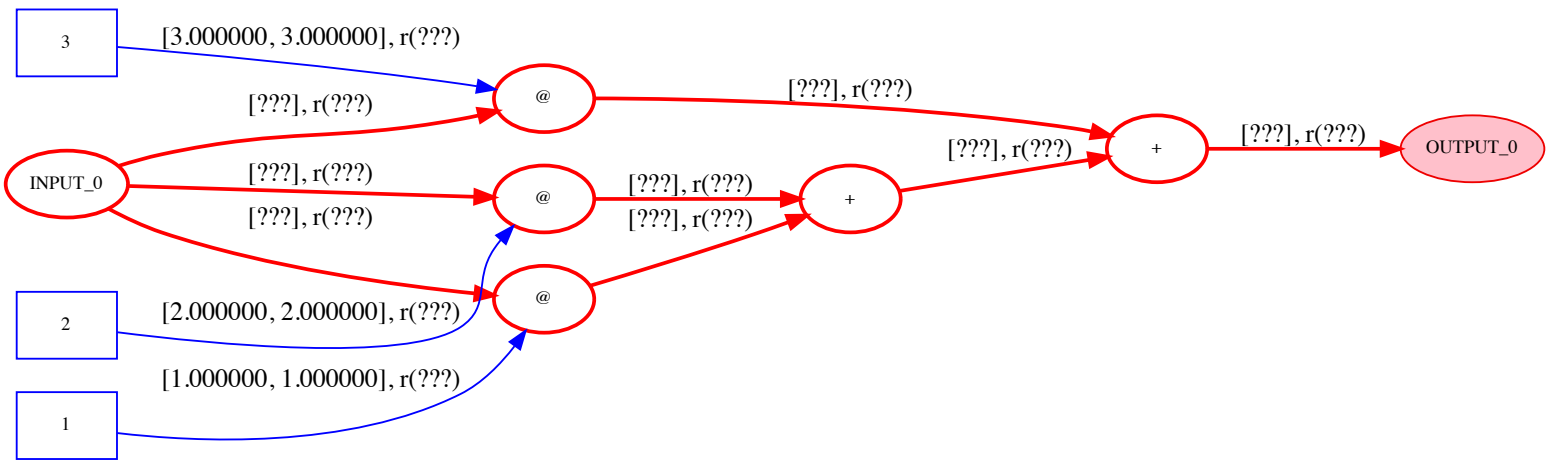
Example:

```
rsub(x,y,z) = y - x*z;
process = rsub(0.5);
```



Symbolic Propagation

The purpose of symbolic propagation is to express the output signals in terms of the input signals:

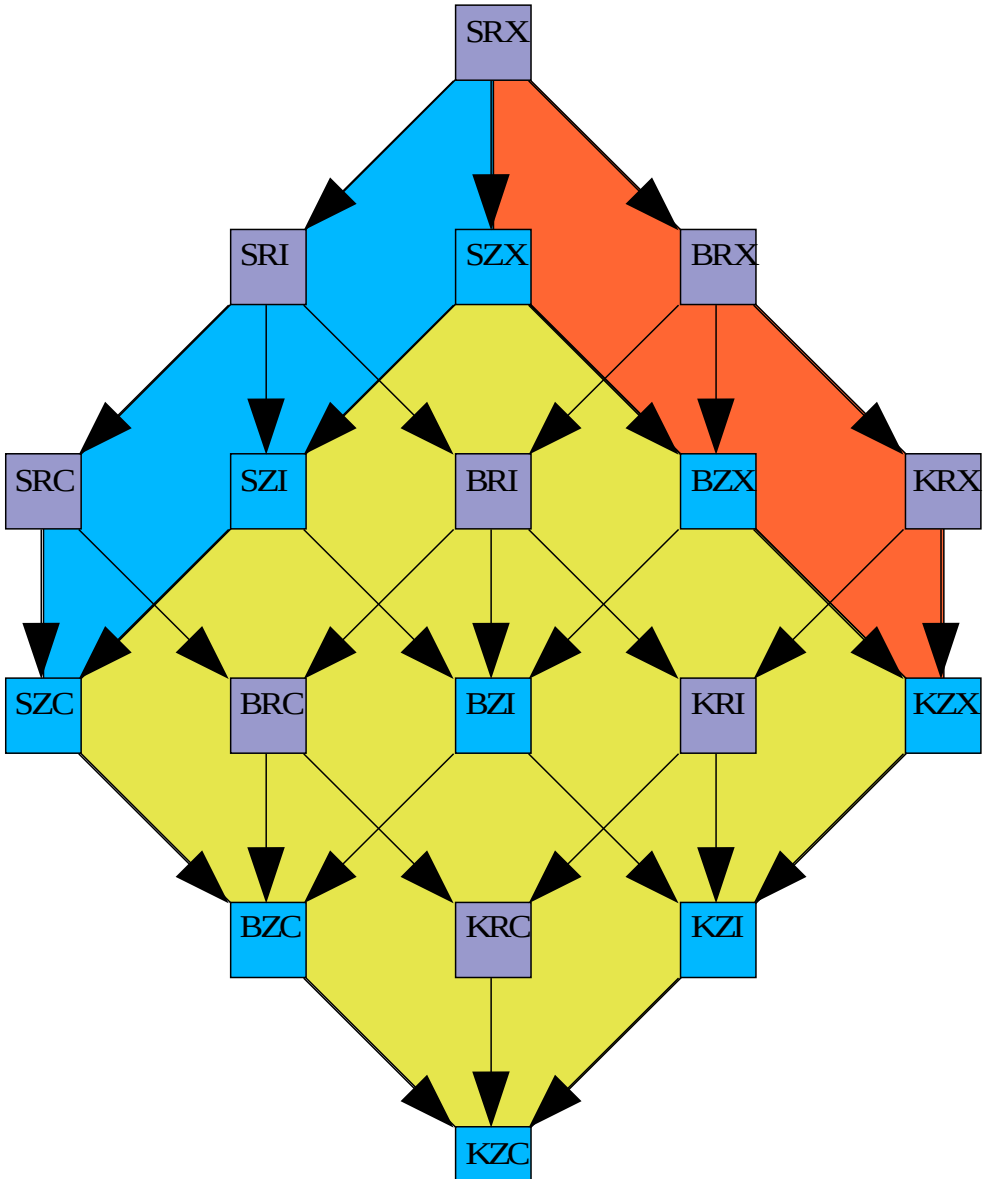


Signals typing

Signal type s = Variability $\times$ Nature $\times$ Calculability

- **Variability** : K (constant) $\subset$ B (bloc/control) $\subset$ S (sample)
- **Nature** : Z (entier) $\subset$ R (réel)
- **Calculability**: C (compilation) $\subset$ I (initialisation) $\subset$ X (execution)

The types form a lattice

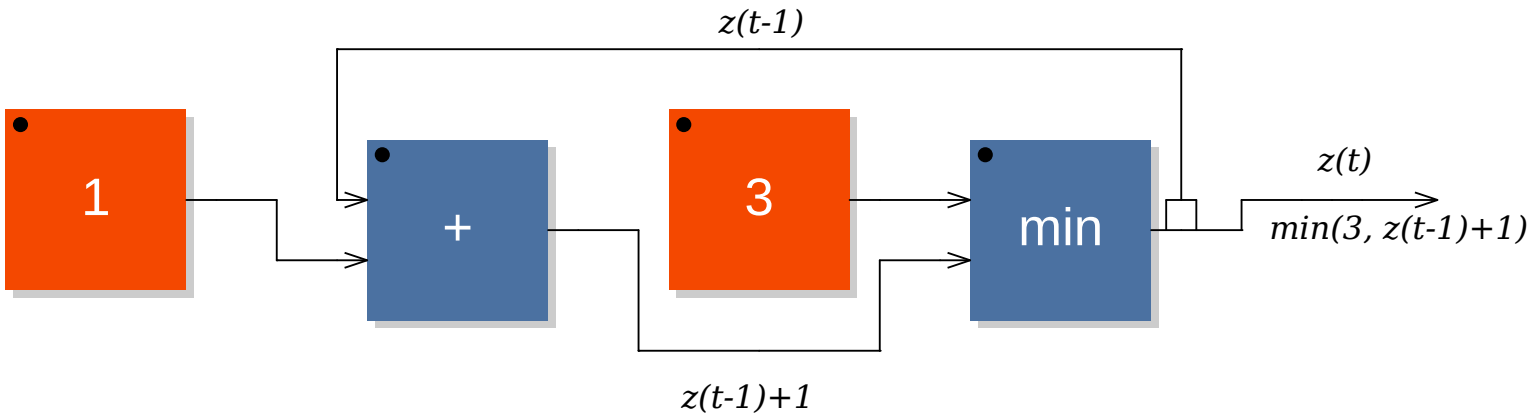


Type of a signal, additional information

- **Vectorability**: $V \subset \widehat{V}$ can be calculated in parallel or not
- **Boolean** : $B \subset \widehat{B}$ represents a Boolean signal or not
- **Interval** : signal values $s(t)$ are contained in the $[l, h]$: $\forall t \in \mathbb{N}, l \leq s(t) \leq h$ interval

Type of signal produced by `(1 : (+ : min(3)) ~ _)`

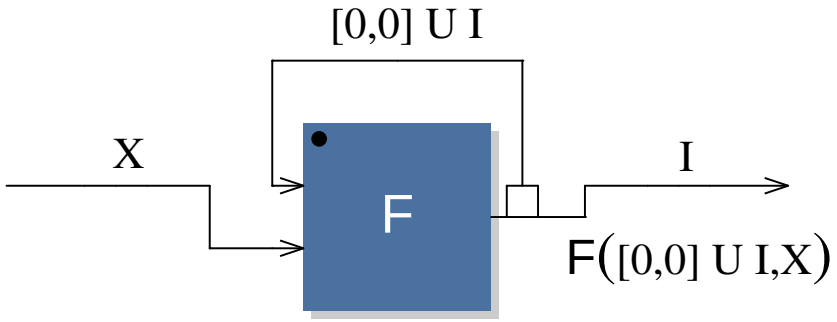
$[\![\texttt{1 : (+ : min(3)) ~ \_}]\!] = () \to z$



Type of $z(t) : SZC\widehat{V}\widehat{B}[1,3]$

Interval of a recursive signal

In reality, we do not really calculate the interval of a recursive signal, we simply return $[-\infty, +\infty]$. What should be done:



- $J_0 = F([0,0], X_0)$, $J_1 = F(J_0, X_1)$, $\ldots$, $J_n = F(J_{n-1}, X_n)$
- $J = \bigcup_{i=0}^{\infty} J_i$, $X = \bigcup_{i=0}^{\infty} X_i$
- $J \subseteq I = F([0,0] \cup I, X)$

Signal processing before translation into FIR

```
Tree L1 = deBruijn2Sym(LS);
typeAnnotation(L1, gGlobal->gLocalCausalityCheck);
SignalPromotion SP;
Tree L1b = SP.mapself(L1);
Tree L2 = simplify(L1b);     // simplify by executing
                             // every computable operation
SignalConstantPropagation SK;
Tree L2b = SK.mapself(L2);
Tree L3 = privatise(L2b);    // un-share tables with
                             // multiple writers
conditionAnnotation(L3);
recursivnessAnnotation(L3); // annotate L3 with
                             // recursivness information
typeAnnotation(L3, true);    // annotate L3 with
                             // type information
sharingAnalysis(L3);         // annotate L3 with sharing count
fOccMarkup = new old_OccMarkup(fConditionProperty);
fOccMarkup->mark(L3); // annotate L3 with occurrences analysis
return L3;
```

Example of normalisation rules

- `s@0` $\rightarrow$ `s`
- `0@d` $\rightarrow$ `0`
- `(k*s)@d` $\rightarrow$ `k*(s@d)`
- `(s/k)@d` $\rightarrow$ `(s@d)/k`
- `(s@n)@m` $\rightarrow$ `s@(n+m)`, if `n` is constant
- `(s+s)` $\rightarrow$ `2*s`
- `(s*s)` $\rightarrow$ `s^2`

Translation of signals into imperative code (FIR: Faust Imperative Representation)

Generic intermediate language before final code generation :

- memory managment: **variables** (`stack/struct/global`), **arrays**, **load/store**
- **arithmetic operators** (unary/binary, external functions)
- control structures: `for`, `while`, `if`, `switch/case`, `select...`
- **data structure** construction
- **functions** creation
- special instructions to **generate the controllers**: building `sliders/buttons/bargraph`

Implementation

Classes to describe and manipulate the FIR:

- notions of:
  - **type**: classe `Typed`
  - **values**: classe `ValueInst`, calculus results
  - **statements**: `StatementInst` class, *side effect* operations
- building of expressions (using the **InstBuilder** class)
- mecanism to **clone** and expression
- mecanism of **visitor** to browse an expression
- files: generator/instructions.hh+cpp

FIR => FIR transformations

Transformations examples:

- renaming or changing the type of variables, example with `stack => struct`
- deletion of unnecessary castings
- functions inlining
- files: generator/fir_to_fir.hh+cpp

Traduction signaux => FIR

The output signals are transformed into FIR expressions with the following classes:

- **CodeContainer** class :
  - progressive filling of the FIR code to generate the DSP structure and the different functions (`init`, `compute`...)
  - subclasses for table generation
- **InstructionsCompiler** class for scalar code generation
- **DAGInstructionsCompiler** class for the generation from the DAG of loops:
  - vector code (loops linked by buffers)
  - vector and parallel code: pragma for OpenMP (C/C++) and Work Stealing Schedule
- files:
  - generator/code_container.hh+cpp
  - generator/instructions_compiler.hh+cpp
  - generator/dag_instructions_compiler.hh+cpp

Compilation: dispatch by signal type

```
ValueInst* InstructionsCompiler::generateCode(Tree sig)
{
    int i; double r;
    Tree c, sel, x, y, z, label, id;
    Tree ff, largs, type, name, file, sf;

    if (getUserData(sig)) {
        return generateXtended(sig);
    } else if (isSigInt(sig, &i)) {
        return generateIntNumber(sig, i);
    } else if (isSigReal(sig, &r)) {
        return generateRealNumber(sig, r);
    } else if (isSigInput(sig, &i)) {
        return generateInput(sig, i);
    } else if {
        ...
    }
}
```

Code generation by the chosen backend

Each backend translates the FIR code into the target language, taking into account its particularities:

- translation from FIR to target language
- possibly uses FIR => FIR operations
- code to generate the structure of the class, module, etc.
- uses the visitor mechanism to convert each FIR expression

Textual backends

Textual backends generate text (an `iostream` in C++) :

- C: data structure and function generation (files in generator/c)
- C++: generation of a class (files in generator/cpp)
- CSharp: generation of a class (files in generator/csharp)
- Rust: generation of a type and methods (files in generator/rust)
- SOUL: generation of a processor (files in generator/soul)
- ...

Others backends

These backends allow to generate code that can then be compiled in memory (LLVM JIT and WASM JIT):

- LLVM IR: generation of an "LLVM module", in the form of data structures in memory, using LLVM libraries (files in generator/llvm)
- WASM: generation of a "WASM module" (files in generator/wasm), in the form of a binary flow, with the help of some complementary intermediate data structures
- ...

Code generation for embedded applications

Some backends have particular generation modes:

- `-os` (one sample) mode with:
  - a `compute` function that computes a single sample
- separation of computations done at `control-rate` and `sample-rate`, in `compute` and `control`

Debugging with the FIR backend

Tool used for debugging FIR and backend implementation:

- textual version of the FIR language :
  - with variable types (`stack, struct, global`)
  - some statistics on the code: size of the DSP, number of operations of each type used (memory access, arithmetic calculations. . .)
- files in generator/fir

Instrumentation with the Interpretation backend

Other backend to generate executable code in memory:

- FIR => Faust Byte Code (FBC) translation
- FBC interpretation virtual machine (with stacks and DSP `integer/real` memory areas)
- possible instrumentation of the code :
  - detection of problematic floats (`NaN`, `INF`...) or integers outside the maximum range, division by zero...
  - incorrect access to memory : test of the correction of the generated code
- files in generator/interp