Présentation du compilateur Faust

Yann Orlarey, Stéphane Letz

1 avril 2021

Les grandes étapes

- Représentations internes basée sur des arbres non mutables et le hash-consing
- Parsing Lex/Yacc
- Evaluation du programme sous la forme d'un circuit de processeurs de signaux
- Propagation symbolique de signaux dans le circuit
- Normalisation et optimisation des signaux
- Typage et calcul d'intervals
- Traduction des signaux en code impératif (FIR)
- Génération du code par le backend choisi

Arbres

Arbres non-mutables, hash-consing, DAG, propriétés mutables.

Propriété des arbres : $t_1 = t_2 \Leftrightarrow M(t_1) = M(t_2)$

- symbols : tlib/symbol.hh+cpp
- nodes : tlib/node.hh+cpp
- trees : tlib/tree.hh+cpp, 2 types de recursivité (de Bruijn+ symbolique)
- constructeurs
- destructurateurs
- propriétés

Arbres : exemple de la composition séquentielle `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);
}
```

Arbres : récursivité *de Bruijn*

```
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;
    }
}
```

Arbres : récursivité *symbolique*

```
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); }
```

Parsing Lex/Yacc

- `parser/faustlexer.l`
- `parser/faustparser.y`
- `libcode.cpp/parseSourceFiles()`
- `parser/sourcereader.hh/SourceReader`
- environnement
- Chargeur récursif, utilisation des url

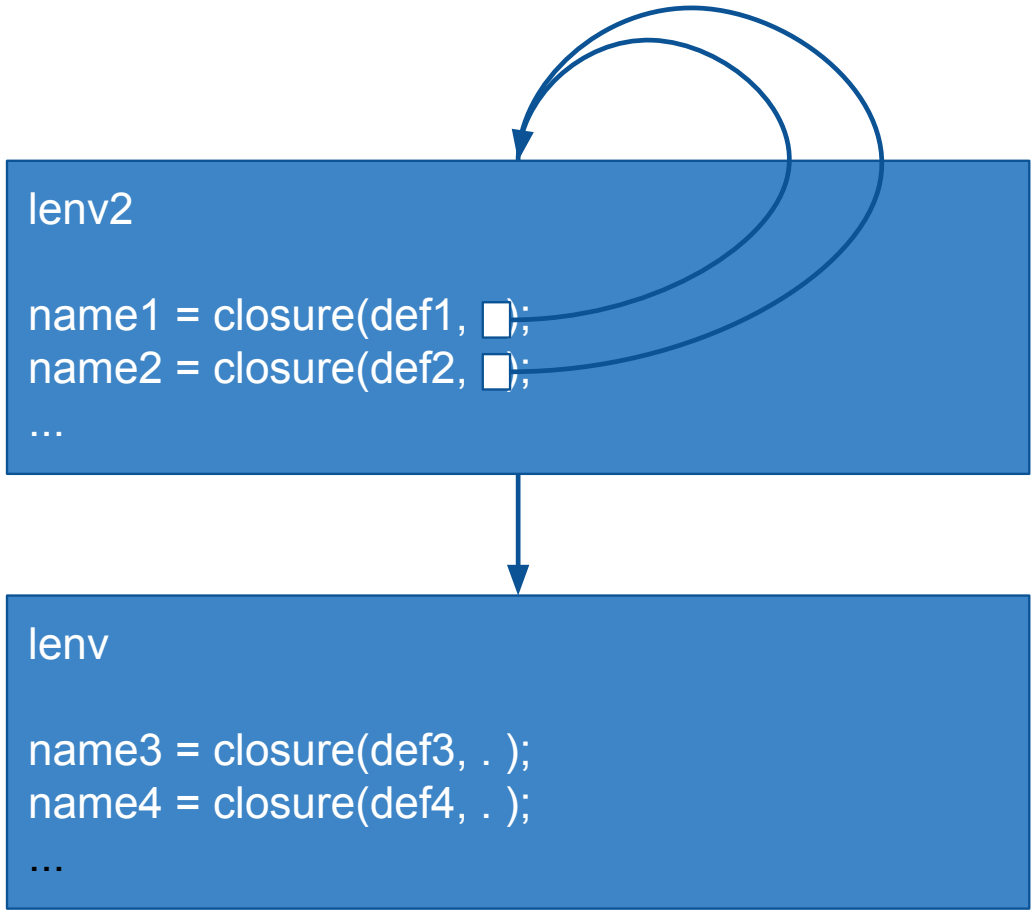
Parsing

Parsing de l'expression `library("math.lib")` :

```
(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;
```

Environnements

Les définitions d'un programme sont organisées en environnements par `pushMultiClosureDefs()` :



Evaluation

Evaluation de la définition de `process` dans l'environnement résultant de la lecture des fichiers sources (voir `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;
}
```

Exemple d'évaluation

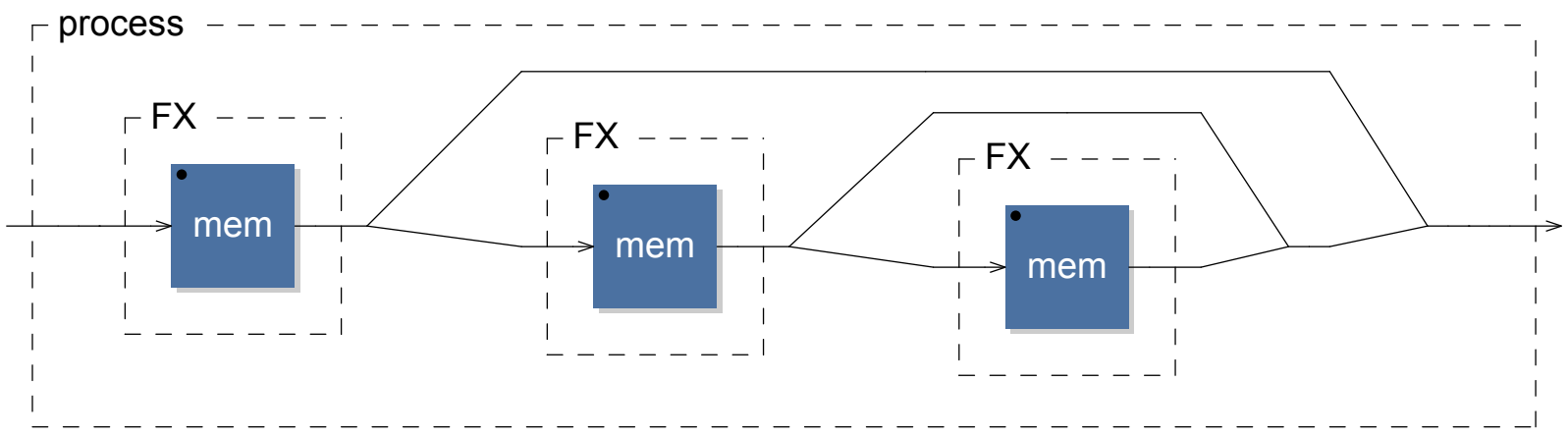
```
repeat(1,f) = f;
repeat(n,f) = f <: _, repeat(n-1,f) :> _;

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

Forme Normale

Le résultat de l'évaluation est un circuit *en forme normale* ou ne subsiste qu'une composition de primitives :

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

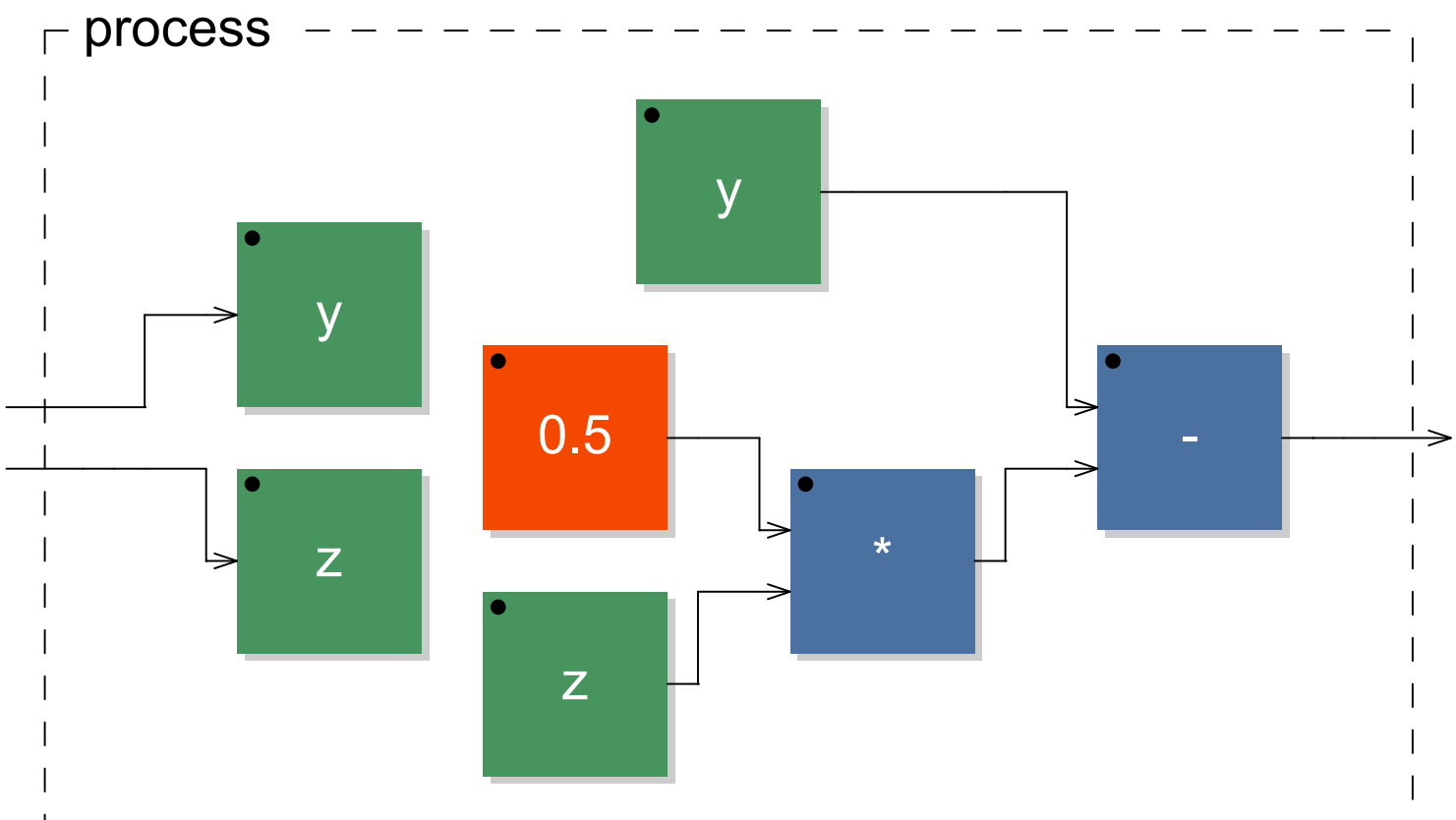
Le diagramme SVG est la représentation graphique (éventuellement hiérarchisée) de la forme normale :



Les abstractions restantes (non appliquées) sont transformées en routage

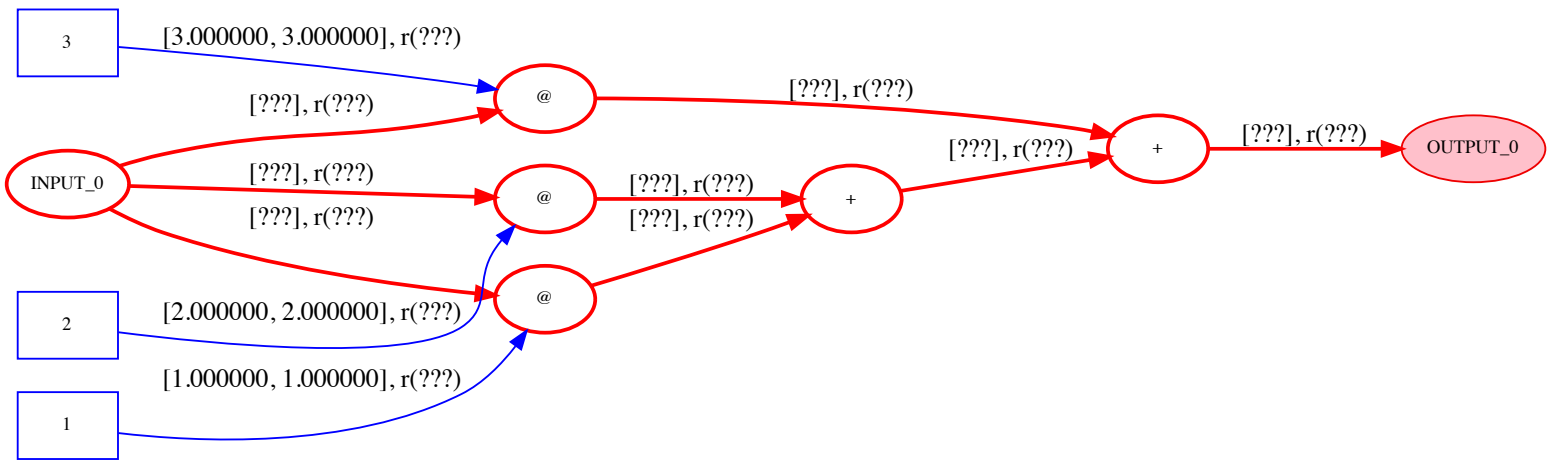
Exemple :

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



Propagation Symbolique

Le but de la propagation symbolique est d'exprimer les signaux de sortie en fonction des signaux d'entrée.

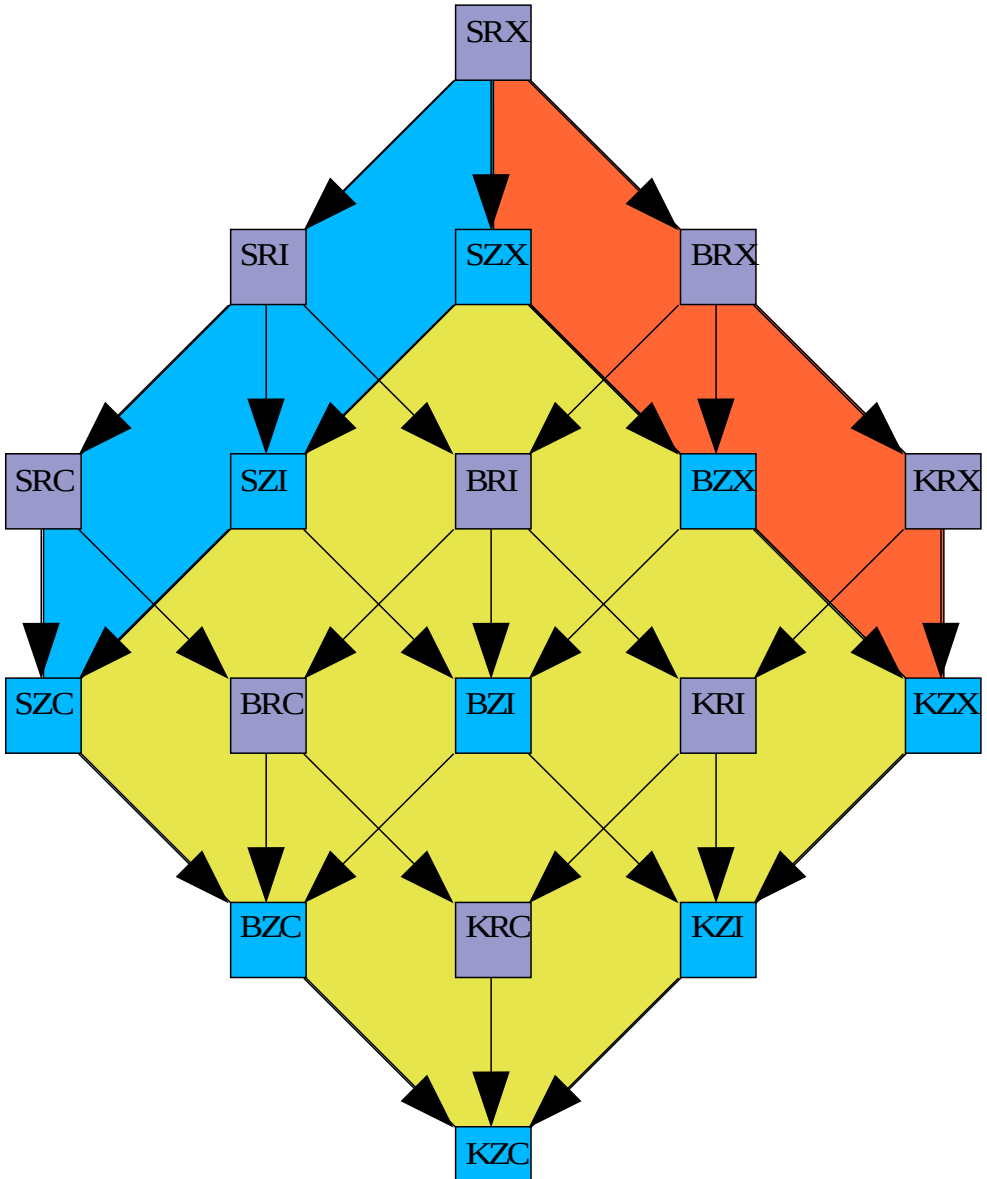


Types des signaux

Type d'un signal s = Variabilité $\times$ Nature $\times$ Calculabilité

- **Variabilité** : K (constant) $\subset$ B (bloc/contrôle) $\subset$ S (sample)
- **Nature** : Z (entier) $\subset$ R (réel)
- **Calculabilité** : C (compilation) $\subset$ I (initialisation) $\subset$ X (exécution)

Les types forment un treillis

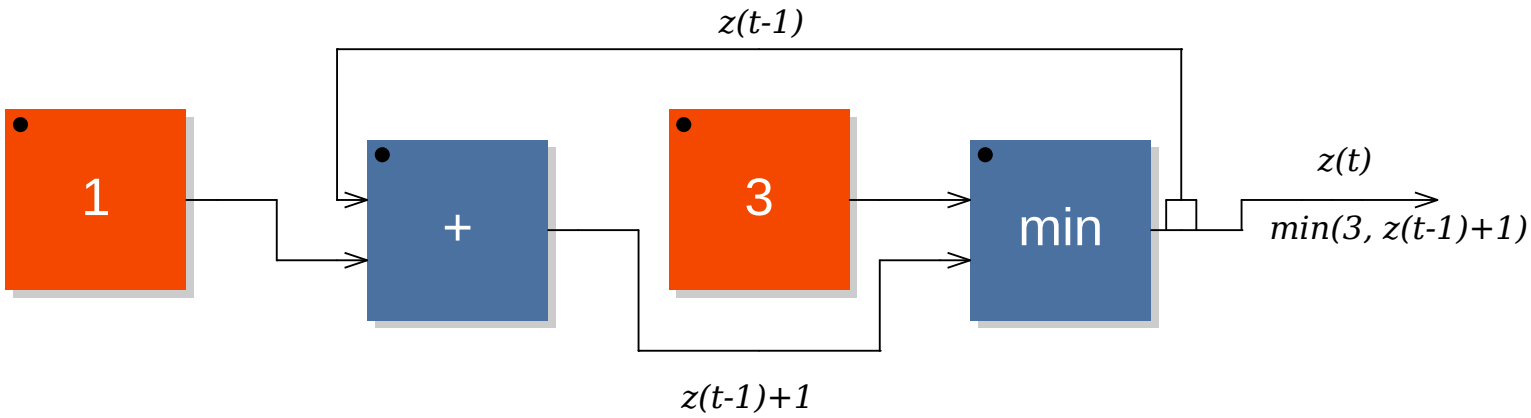


Type d'un signal, informations additionnelles

- **Vectorabilité** : $V \subset \widehat{V}$ peut être calculé en parallèle ou pas
- **Booléen** : $B \subset \widehat{B}$ représente un signal booléen ou pas
- **Intervalle** : les valeurs du signal $s(t)$ sont contenues dans l'intervalle $[l, h]$: $\forall t \in \mathbb{N}, l \leq s(t) \leq h$

Type du signal produit par `(1 : (+ : min(3)) ~ _)`

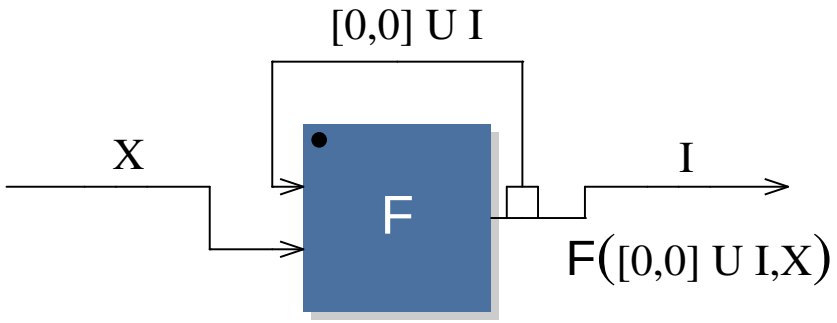
$[\![\texttt{1 : (+ : min(3)) ~ \_}]\!] = () \to z$



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

Intervalle d'un signal récursif

En réalité, on ne calcule pas réellement l'intervalle d'un signal récursif, on renvoie simplement $[-\infty, +\infty]$. Ce qu'il faudrait faire :



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

Traitement des signaux avant la traduction en 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;
```

Exemple de règles de normalisation

- `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)`, si `n` est constant
- `(s+s)` $\rightarrow$ `2*s`
- `(s*s)` $\rightarrow$ `s^2`

Traduction des signaux en code impératif (FIR: Faust Imperative Representation)

Langage générique intermédiaire avant la génération du code final :

- gestion mémoire: **variables** (`stack/struct/global`), **tableaux**, **lecture/écriture**
- **opérations arithmétiques** (unaires/binaires), fonctions externes
- structure de contrôle : `for`, `while`, `if`, `switch/case`, `select...`
- création de **structures de données**
- création de **fonctions**
- instructions spéciales pour **générer les controlleurs** : construction de `sliders/buttons/bargraph`

Implémentation

Classes pour décrire et manipuler le FIR:

- notions de:
  - **type** : classe `Typed`
  - **values** : classe `ValueInst`, résultat des calculs
  - **statements** : classe `StatementInst`, opérations à *effet de bord*
- construction d'expressions (avec la classe **InstBuilder**)
- mécanisme de **clonage** d'une expression
- mécanisme de **visiteur** pour parcourir une expression
- fichiers : generator/instructions.hh+cpp

Transformations FIR => FIR

Exemples de transformations:

- renomage ou changement de type de variables, example avec `stack => struct`
- suppressions de cast inutiles
- inlining de fonctions
- fichiers : generator/fir_to_fir.hh+cpp

Traduction signaux => FIR

Les signaux de sortie sont transformés en expressions FIR avec les classes suivantes:

- classe **CodeContainer** :
  - remplissage progressif du code FIR pour générer la structure DSP et les différentes fonctions (`init`, `compute`...)
  - sous-classes pour la génération des tables
- classe **InstructionsCompiler** pour la génération de code scalaire
- classe **DAGInstructionsCompiler** pour la génération à partir du DAG de boucles :
  - code vectoriel (boucles reliées par des buffers)
  - code vectoriel et parallèle : pragma pour OpenMP (C/C++) et Work Stealing Scheduler
- fichiers :
  - generator/code_container.hh+cpp
  - generator/instructions_compiler.hh+cpp
  - generator/dag_instructions_compiler.hh+cpp

Compilation : dispatch par type de signal

```
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 {
        ...
    }
}
```

Génération du code par le backend choisi

Chaque backend traduit le code FIR dans le langage cible, en tenant compte de ses particularités:

- traduction du FIR vers le langage cible
- utilise éventuellement des opérations FIR $=>$ FIR
- code pour générer la structure de la classe, du module, etc.
- utilise le mécanisme de visiteur pour convertir chaque expression FIR

Backends textuels

Les backends textuels génèrent du texte (un `iostream` en C++) :

- C : génération de structure de données et fonctions (fichiers dans generator/c)
- C++ : génération d'une classe (fichiers dans generator/cpp)
- CSharp : génération d'une classe (fichiers dans generator/csharp)
- Rust : génération d'un type et de méthodes (fichiers dans generator/rust)
- SOUL : génération d'un processor (fichiers dans generator/soul)
- ...

Autres backends

Ces backends permettent de générer du code ensuite compilable en mémoire (LLVM JIT et WASM JIT) :

- LLVM IR : génération d'un « module LLVM » , sous la forme de structures de données en mémoire, à l'aide des librairies LLVM (fichiers dans generator/llvm)
- WASM : génération d'un « module WASM » (fichiers dans generator/wasm), sous la forme d'un flux binaire, à l'aide de quelques structure de données intermediaires complémentaires
- ...

Génération de code pour l'embarqué

Certains backends ont des modes de génération particuliers:

- mode `-os` (one sample) avec :
  - fonction `compute` qui calcule un seul échantillon
  - séparation des calculs faits au `control-rate` et au `sample-rate`, dans `compute` et `control`

Déboggage avec le backend FIR

Outil utilisé pour le déboggage du FIR et de l'implémentation des backends :

- version textuelle du langage FIR :
  - avec type des variables (`stack`, `struct`, `global`)
  - quelques statistiques sur le code : taille du DSP, nombre d'opérations de chaque type utilisées (accès mémoire, calculs arithmétiques. . .)
- fichiers dans generator/fir

Instrumentation avec le backend d'Interprétation

Autre backend pour générer du code exécutable en mémoire:

- traduction FIR => Faust Byte Code (FBC)
- machine virtuelle d'interprétation du FBC (avec piles et zones mémoires DSP `integer/real`)
- instrumentation du code possible :
  - détection de calculs flottants problématiques (`NaN`, `INF`...) ou entiers en dehors de l'intervalle maximum, division par zéro...
  - accès incorrect à la mémoire : test de la correction du code généré
- fichiers dans generator/interp