Type inference in the multirate audio DSP language Faust

Pierre Beauguittte

Centre de recherche en informatique - MINES ParisTech

SYNCHRON 2012
FAUST : Functional AUdio STream

Language developed at GRAME (Lyon) since 2003.


Compiled language, real-time audio applications.

Audio synthesis, treatments; interactive applications.

Work at sample-level (typically 44.1 kHz).
Plan

1. The Faust language
2. Vector extension
3. Type inference
The Faust language

Vector extension

Type inference

Overview

Domain-specific language, audio digital signal processing.

- synchronous
- purely functional
- textual block-diagram description
- statically typed

```
random = +(12345)~*(1103515245);
noise = random/2147483647.0;
process = noise * vslider("vol",0,0,1,0.1);
```
Domain-specific language, audio digital signal processing.

- synchronous
- purely functionnal
- textual block-diagram description
- statically typed

\[
\text{random} = +(12345)\times(1103515245);
\]
\[
\text{noise} = \text{random}/2147483647.0;
\]
\[
\text{process} = \text{noise} \times \text{vslider("vol",0,0,1,0.1)};
\]
- Automatic generation of optimized C++ code
- Online compiler: http://faust.grame.fr/
- Multi-target compilation

Diagram:

- Faust code
  - Standalone application
  - Max/MSP or PureData external
  - Web application
  - VST plug in
  - iOS application
Block diagrams are built using 5 composition operators:

\[
y(t) = \left[ \frac{x_1(t)}{x_2(t)} \prod \right]
\]

/ : floor

Type inference in multirate Faust
Pierre Beauguette
Block diagrams are built using 5 composition operators:

\[ y(t) = \left\lfloor \frac{x_1(t)}{x_2(t)} \right\rfloor \]
\[ y(t) = (10, x(t)) \]
Block diagrams are built using 5 composition operators:

\[
\begin{align*}
\frac{x_1(t)}{x_2(t)} & : \text{floor} \\
y(t) &= \left\lfloor \frac{x_1(t)}{x_2(t)} \right\rfloor \\
10, \_ & \quad 10, x(t) \\
\_, 2 \_ : +, * & \quad y(t) = (x(t) + 2, 2x(t))
\end{align*}
\]
\[ y(t) = (x_1(t) + x_3(t)) \ast (x_2(t) + x_4(t)) \]
\[ y(t) = (x_1(t) + x_3(t)) \cdot (x_2(t) + x_4(t)) \]

\[ \begin{align*}
  y(0) &= x(0) + \cos(0) \\
  y(t) &= x(t) + \cos(y(t - 1))
\end{align*} \]
Faust language

High-level functional language with lambdas, libraries, pattern-matching, infix notations, local environments...

```
fact(n) = case {
    (0) => 1;
    (n) => (n, fact(n-1)) : *
};
```

```
q(x,y) = floor(x/y);    // stands for x,y : / : floor
```

```
mix = \(n).(par(i,n,_) :> _);
```
Expressiveness

Faust is Turing-complete.

The current version is monorate; wires carry only scalar signals.

We need

- different rates
- more complex data structures (vectors, matrices...)

to deal with multirate signal processing or spectral analyses.
Types

BasicType = \{\text{Int, Float}\}
Interval = \mathbb{R}^\omega \times \mathbb{R}^\omega \quad (\mathbb{R}^\omega = \mathbb{R} \cup \{-\omega, \omega\})
Frequency = \mathbb{Q}_+$
Types

BasicType = \{\text{Int, Float}\}
Interval = \mathbb{R}^\omega \times \mathbb{R}^\omega \quad (\mathbb{R}^\omega = \mathbb{R} \cup \{-\omega, \omega\})
Frequency = \mathbb{Q}_+

\tau \in \text{Type} = \text{BasicType} \times \text{Interval} \quad \text{e.g. Float}[0, 1]
\quad | \quad \mathbb{N}^* \times \text{Type} \quad \text{e.g.} \quad \text{vector}_n(\tau)

A signal has a type \tau and a frequency \(f\), written \(\tau^f\)
Types

BasicType = \{ \text{Int, Float} \}
Interval = \mathbb{R}^\omega \times \mathbb{R}^\omega \quad (\mathbb{R}^\omega = \mathbb{R} \cup \{-\omega, \omega\})
Frequency = \mathbb{Q}_+

\tau \in \text{Type} = \text{BasicType} \times \text{Interval} \quad \text{e.g. } \text{Float}[0, 1]
| \quad \mathbb{N}^* \times \text{Type} \quad \text{e.g. } \text{vector}_n(\tau)

A signal has a type \( \tau \) and a frequency \( f \), written \( \tau^f \)

Subtyping rules:

\[ [x, y] \subset [x', y'] \Rightarrow b[x, y]^f \subset b[x', y']^f \]

\[ \text{Int}[x, y]^f \subset \text{Float}[x, y]^f \]

\[ \tau^0 \subset \tau^f \]

\[ \tau \subset \tau' \Rightarrow \text{vector}_n(\tau)^f \subset \text{vector}_n(\tau')^f \]
Initial environment

Associates to predefined identifiers their input and output types.

\[
T(\_ ) = \Lambda f : \text{Rate.}\tau : \text{Type.} (\tau^f) \rightarrow (\tau^f) \\
T(0) = \Lambda f : \text{Rate.}(\_ ) \rightarrow (\text{Int}[0,0]^0) \\
T(+) = \Lambda f : \text{Rate.}\tau : \text{Type.}\tau' : \text{Type.} (\tau^f, \tau'^f) \rightarrow (\tau + \tau')^f
\]

Binary operations are well formed if:

\[
\exists \bar{\tau} / (\tau \subset \bar{\tau} \land \tau' \subset \bar{\tau}).
\]
Vector primitives

\[ T(\text{vectorize}) = \Lambda f : \text{Rate.}\tau : \text{Type.n : } \mathbb{N}^*. \\
(\tau^f, \text{Int}[n, n]^0) \rightarrow (\text{vector}_n(\tau)^{f/n}) \]

\[ T(\text{serialize}) = \Lambda f : \text{Rate.}\tau : \text{Type.n : } \mathbb{N}^*. (\text{vector}_n(\tau)^f) \rightarrow (\tau^{n.f}) \]
\[ T(\#) = \Lambda f : \text{Rate.} \tau : \text{Type.} \tau' : \text{Type.} n : \mathbb{N}^*. n' : \mathbb{N}^*. \]
\[ (\text{vector}_n(\tau)^f, \text{vector}_{n'}(\tau')^f) \rightarrow (\text{vector}_{n+n'}(\tau \sqcup \tau')^f) \]

\[ T(\square) = \Lambda f : \text{Rate.} \tau : \text{Type.} n : \mathbb{N}^*. (\text{vector}_n(\tau)^f, \text{Int}[0, n-1]^f) \rightarrow (\tau^f) \]
Semantic rules

(i) \[ T(I) = \forall l. z \rightarrow z' \]
\[ \forall (x, S) \in l, \quad l' [l^{-1}(x, S)] \in S \]
\[ T \vdash I : (z \rightarrow z')[l'/l] \]

(\subset)
\[ T \vdash E : z \rightarrow z' \]
\[ z_1 \subset z \]
\[ z' \subset z'_1 \]
\[ T \vdash E : z_1 \rightarrow z'_1 \]

(:,)
\[ T \vdash E_1 : z_1 \rightarrow z'_1 \]
\[ T \vdash E_2 : z_2 \rightarrow z'_2 \]
\[ T \vdash E_1, E_2 : z_1 || z_2 \rightarrow z'_1 || z'_2 \]

(:)
\[ T \vdash E_1 : E_2 : z_1 \rightarrow z'_2 \]
\[ T \vdash E_2 : z'_1 \rightarrow z'_2 \]
\[ T \vdash E_1 : E_2 : z_1 \rightarrow z'_2 \]
Polymorphism and overloading

- All primitives are polymorphic due to abstractions in type schemes
- :> adds vector signals pointwise
- Overloading of arithmetic operators
A Faust process is a well-typed expression such that all its I/O signals are scalar and of non-zero frequency.

The non-zero frequency condition ensures that all vector dimensions are known at compile time.
The Faust language
Vector extension
Type inference

Goal

Static type inference of annotation free code.

\[
\text{vectorize}(4), \text{vectorize}(2) : \# : \text{serialize}
\]

\[
\left( \tau^4, \tau'^2 \right) \rightarrow \left( (\tau \sqcup \tau')^6 \right)
\]
Type representation, environment

Isomorphic representation of types:

\[ t : \text{Type} \rightarrow \text{Range} \times \text{Dimension} \]

\[ d \in \text{Dimension} = \text{Scalar} \quad (\text{for } b[x, y]) \]
\[ | \quad n :: d' \quad (\text{for } \text{vector}_n(\tau)) \]

For instance, \( t(\text{vector}_3(\text{vector}_2(\text{Int}[0, 1]))) = (\text{Int}[0, 1], [3, 2]). \)
Isomorphic representation of types:

\[ t : \text{Type} \rightarrow \text{Range} \times \text{Dimension} \]

\[ d \in \text{Dimension} = \text{Scalar} \quad \text{(for b}[x, y]) \]
\[ \mid n :: d' \quad \text{(for vector}_n(\tau)) \]

For instance, \( t(\text{vector}_3(\text{vector}_2(\text{Int}[0, 1]))) = (\text{Int}[0, 1], [3, 2]). \)

- \( \rightarrow (r, d, f) \rightarrow (r, d, f), \emptyset \)
- \( 0 \rightarrow () \rightarrow (\text{Int}[0, 0], \text{Scalar}, 0), \emptyset \)
Isomorphic representation of types:

\[ t : \text{Type} \rightarrow \text{Range} \times \text{Dimension} \]

\[ d \in \text{Dimension} = \begin{cases} \text{Scalar} & \text{(for } b[x, y]) \\ n :: d' & \text{(for } \text{vector}_n(\tau)) \end{cases} \]

For instance, \( t(\text{vector}_3(\text{vector}_2(\text{Int}[0, 1]))) = (\text{Int}[0, 1], [3, 2]) \).

\[
\begin{align*}
_ - & \mapsto (r, d, f) \rightarrow (r, d, f), \emptyset \\
0 & \mapsto () \rightarrow (\text{Int}[0, 0], \text{Scalar}, 0), \emptyset \\
# & \mapsto (r, n :: d, f), (r', n' :: d', f) \\
& \rightarrow (r \sqcup r', (n + n') :: d, f), \{d = d'\}
\end{align*}
\]
Algorithm: constraint generation

\[
\text{type}(E, L_0) = \text{match } E \text{ with } \\
I \mapsto \text{New } (\text{Env } (I), L_0) \\
E_1, E_2 \mapsto (I_1 \parallel I_2 \rightarrow O_1 \parallel O_2), C_1 \cup C_2, L_2 \\
E_1 : E_2 \mapsto (I_1 \rightarrow O_2), C_1 \cup C_2 \cup \text{subbeam}(O_1, I_2), L_2 \\
\ldots
\]

where \((I_i \rightarrow O_i, C_i, L_i) = \text{type}(E_i, L_{i-1})\), \\
\text{New} creates a new instance of } Env(I) \text{ with fresh variables,} \\
L_i \text{ is the set of used variables.}
Constraints reduction

- Dimension equalities and frequency relations are reduced to numerical equalities and substitutions with inference systems:

\[
\{ d_i = d \} \cup D; \mathcal{N}; S \Rightarrow D[d/d_i]; \mathcal{N}; S[d/d_i] \cup \{ d_i \mapsto d \}
\]

\[
\{ n :: d = n' :: d' \} \cup D; \mathcal{N}; S \Rightarrow \{ d = d' \} \cup D; \mathcal{N} \cup \{ n = n' \}; S
\]

\[
\{ \text{Scalar} = n :: d \} \cup D; \mathcal{N}; S \Rightarrow \text{fail}
\]

- Range relations can lead to
  - static computation of vector dimensions
  - static verification of arithmetic relations
  - dynamic clipping of signals
Correctness

Theorem (Soundness)

Let $E$ be a Faust expression, and $(I \rightarrow O, C, L) = type(E, \emptyset)$. Then, if $\mathcal{M}$ is a model defined on $L$ such that $\mathcal{M} \models C$, then $T \vdash E : t^{-1}(\mathcal{M}(I \rightarrow O))$.

Theorem (Completeness)

Let $E$ be a Faust expression, such that $T \vdash E : z \rightarrow z'$. Then $type(E, \emptyset) = (I \rightarrow O, C, L)$, and there exists a model $\mathcal{M}$ defined on $L$ such that $\mathcal{M} \models C$ and $t^{-1}(\mathcal{M}(I \rightarrow O)) \subset z \rightarrow z'$. 
Conclusion

- Static inference of rate relations and vector dimensions. OCaml prototype.
  How to gain precision on data signal types?

- Expressive power of Faust
  Is the vector extension well suited for DSP algorithms?
  Study cases with IRCAM
Type inference in the multirate audio DSP language Faust

Pierre Beauguittte

Centre de recherche en informatique - MINES ParisTech

SYNCHRON 2012