



A Constraint-Solving Approach to Faust Program Type Checking

Constraint Programming Meets Verification 2014 Workshop

Imré Frotier de la Messelière¹, Pierre Jouvelot¹, Jean-Pierre Talpin²

¹MINES ParisTech, PSL Research University

²INRIA

September 8, 2014

Contents

- 1 Faust program type checking
- 2 Constraint-solving approach
- 3 Handling the multirate version of Faust
- 4 Type checking examples and results

- 1 Faust program type checking
- 2 Constraint-solving approach
- 3 Handling the multirate version of Faust
- 4 Type checking examples and results

Faust program type checking

Motivation

- Specification and implementation of a new type inference algorithm for Faust
- Inspiration from:
 - ▶ Hindley-Milner's algorithm W [3]
 - ▶ the algebraic reconstruction approach of Jouvelot and Gifford [1]
- Formally proven static typing system \implies Better reliability and efficiency

Constraint-programming approach

- Use of constraints as the foundation of the whole typing process
- Contrary to more standard approaches adopting techniques based on substitutions and principal types
- Creation of large and multi-sorted constraint systems that will need to be processed efficiently

Faust program type checking

Type syntax for a Faust monorate expression

expression_type ::= (beam_type,beam_type)

beam_type ::= signal_type list

signal_type ::= base_type [x,x]

base_type ::= int | float

$x \in \mathbb{Z}$

Type examples

1 \Rightarrow ((), (int [-2,2]))

2 \Rightarrow ((), (int [0,3]))

+ \Rightarrow ((int [-20,20], int [-20,20]), (int [-40,40]))

1 , 2 : + \Rightarrow ((), (int [-40,40]))

Faust program type checking

Typing rules : \implies Type specification for programmers

$$\frac{T(\mathbf{I}) = \Lambda l.(z, z') \quad \forall(x, S) \in l . \quad l'(x) \in S}{T \vdash \mathbf{I} : (z, z')[l'/l]} \quad \frac{T \vdash E_1 : (z_1, z'_1) \quad T \vdash E_2 : (z'_1, z'_2)}{T \vdash E_1 : E_2 : (z_1, z'_2)} \quad \frac{\begin{array}{c} T \vdash E : (z, z') \\ z' \subset z'_1 \\ z_1 \subset z \end{array}}{T \vdash E : (z_1, z'_1)}$$

$$\frac{\begin{array}{c} T \vdash E_1 : (z_1, z'_1) \\ T \vdash E_2 : (z_2, z'_2) \end{array}}{T \vdash E_1, E_2 : (z_1 \| z_2, z'_1 \| z'_2)} \quad \frac{\begin{array}{c} T \vdash E_1 : (z_1, z'_1) \\ T \vdash E_2 : (z_2, z'_2) \\ z_2 = z'[1, |z_2|] \\ z'_2 = z_1[1, |z'_2|] \end{array}}{T \vdash E_1 \sim E_2 : (z_1[|z'_2| + 1, |z_1|], \widehat{z'})}$$

Faust program type checking

$$\frac{T \vdash E_1 : (z_1, z'_1) \quad T \vdash E_2 : (z_2, z'_2) \quad z'_1 \prec z_2}{T \vdash E_1 <: E_2 : (z_1, z'_2)}$$

$z'_1 \prec z_2 = d'_1 d_2 \neq 0 \text{ and}$
 $\text{mod}(d_2, d'_1) = 0 \text{ and}$
 $\|_{1, d_2, d'_1} \lambda i. z'_1 = z_2$

$$\frac{T \vdash E_1 : (z_1, z'_1) \quad T \vdash E_2 : (z_2, z'_2) \quad z'_1 \succ z_2}{T \vdash E_1 :> E_2 : (z_1, z'_2)}$$

$z'_1 \succ z_2 = d'_1 d_2 \neq 0 \text{ and}$
 $\text{mod}(d'_1, d_2) = 0 \text{ and}$
 $\sum_{i \in [0, d'_1/d_2 - 1]} z_1[1 + id_2, (i + 1)d_2] = z_2$

Faust program type checking

Type checking overview

Algorithm in two parts:

- a classic type inference algorithm, coupled with the generation of typing constraints
- a solver (1) to determine if the resulting constraints system is decidable and (2) to provide a mapping yielding the type of Faust expressions

Implementation

- First prototype in OCaml
- Rewriting in C++ \implies Inclusion within the current Faust compiler
- Based on the type checking algorithm:
 - ▶ Input: Faust expression
 - ▶ Output: type of the Faust expression or “fail”

- 1 Faust program type checking
- 2 Constraint-solving approach
- 3 Handling the multirate version of Faust
- 4 Type checking examples and results

Constraint-solving approach

Origin of constraints

- Environment T: mapping of Faust identifiers to their types
- Identifiers' types plugged into the typing rules

Constraints implementation

- Type templates with type variables in the environment
 $+ : ((\text{int } [a_1, b_1], \text{int } [a_2, b_2]), (\text{int } [a_1+a_2, b_1+b_2]))$
- Templates implemented by replacing type variables by actual values or unification variables (buffer values)
 $1, 1 : + \implies + : ((\text{int } [-1, 1], \text{int } [-1, 1]), (\text{int } [-2, 2]))$
- Unification variables = variables for constraints
- Different possible instances, based on subtyping:
 $1, 10 : + \implies + : ((\text{int } [-1, 1], \text{int } [-10, 10]), (\text{int } [-11, 11]))$

Constraint-solving approach

Predicates syntax

$p \in P ::= \text{true} \mid e \ b \ e$

$e \in E ::= i \mid o_1 \ e \mid e \ o_2 \ e$

$b \in B ::= = \mid < \mid \leq \mid > \mid \geq$

$o_1 \in O_1 ::= \sin \mid \cos \mid \dots$

$o_2 \in O_2 ::= + \mid - \mid \dots$

$i \in I$

Constraints syntax

$c \in C ::= (p \text{ list}, i \text{ list}) \mid c \cup c$

where,

for $c = (ps, is)$ and $c' = (ps', is') \in C$, $c \cup c' = (ps @ ps', is @ is')$

Constraint-solving approach

Constrained types

- $\text{constrained_type} ::= (\text{expression_type}, c)$
- Result of the constraint generation part of the type checking algorithm
- Solver input = c
- Solver output = Mapping m from unification variables to values
- Application of m to expression_type
 \implies Type (Global result of the algorithm)

Constraint-solving approach

Solver

- Solving handled by existing solvers, using SMT-LIB as a common representation framework for constraints
- Currently using Z3
- Possibility to design a lighter solver, only using theories involved in the algorithm?
- Output = mapping of unification variables to values

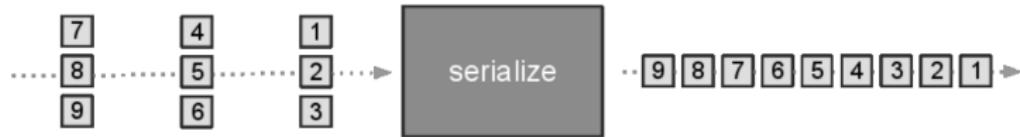
- 1 Faust program type checking
- 2 Constraint-solving approach
- 3 Handling the multirate version of Faust
- 4 Type checking examples and results

Handling the multirate version of Faust

- **vectorize:** Input signal at rate $f \Rightarrow$ Output signal at rate f/n



- **serialize:** Input signal at rate $f \Rightarrow$ Output signal at rate $n \times f$



- **[]:** element access
- **#:** concatenation

Handling the multirate version of Faust

Type syntax for a Faust multirate expression

expression_type ::= (beam_type, beam_type)

beam_type ::= signal_type list

signal_type ::= faust_type^{rate}

faust_type ::= base_type [x,x] | vector_n(faust_type)

base_type ::= int | float

rate $\in \mathbb{Q}^+$

$x \in \mathbb{Z}$

$n \in \mathbb{N}$

Handling the multirate version of Faust

Input sample rate = 44000 Hz:

1 , 2 : vectorize $\Rightarrow (((), (\text{vector}_2(\text{int } [-1,1])^{22000}))$

1 , 2 : vectorize : serialize $\Rightarrow (((), (\text{int } [-1,1]^{44000}))$

1 , 2 : vectorize , 1 :[] $\Rightarrow (((), (\text{int } [-1,1]^{22000}))$

(1,2 : vectorize) , (6,3 : vectorize) :# $\Rightarrow (((), (\text{vector}_5(\text{int } [-10,10])^{22000}))$

1 , 2 : vectorize , 3 : vectorize $\Rightarrow (((), (\text{vector}_3(\text{vector}_2(\text{int } [-1,1])))^{22000}))$

Handling the multirate version of Faust

Additional environment entries in T:

$$T(\text{vectorize}) = (\tau^f, \text{int}[n, n]^{f'}) \longrightarrow (\text{vector}_n(\tau)^{f/n})$$

$$T(\text{serialize}) = (\text{vector}_n(\tau)^f) \longrightarrow (\tau^{f \times n})$$

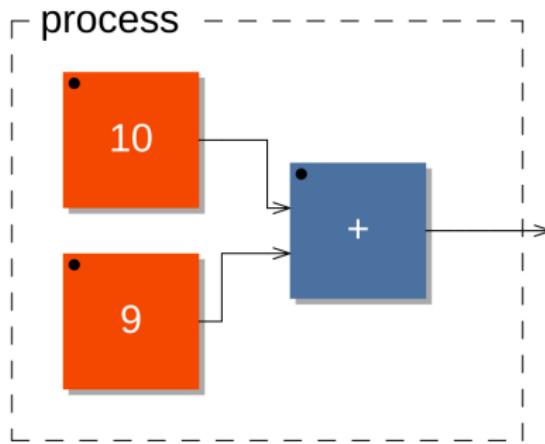
$$T([\]) = (\text{vector}_n(\tau)^f, \text{int}[0, n - 1]^f) \longrightarrow (\tau^f)$$

$$T(\#) = (\text{vector}_m(\tau)^f, \text{vector}_n(\tau)^f) \longrightarrow (\text{vector}_{m+n}(\tau)^f)$$

- 1 Faust program type checking
- 2 Constraint-solving approach
- 3 Handling the multirate version of Faust
- 4 Type checking examples and results

Type checking examples and results

```
process = 10,9:+ ;
```



Constrained type = (

Type: (((),((uv13[uv8+uv11,uv9+uv12])^uv14)),

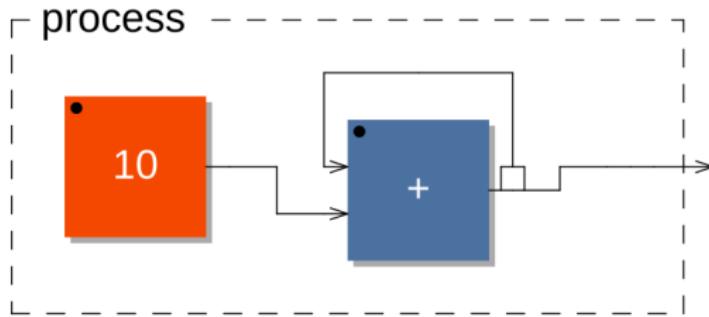
Constraint:

((uv1<=10,uv2>=10,uv4<=9,uv5>=9,uv3==uv14,Int==uv7,uv1>=uv8,uv2<=uv9,
uv6==uv14,Int==uv10,uv4>=uv11,uv5<=uv12),
(uv1,uv2,uv3,uv4,uv5,uv6,uv7,uv8,uv9,uv10,uv11,uv12,uv13,uv14)))

Type = (((),((Int[19,19])^1))

Type checking examples and results

```
process = 10:+~_ ;
```



```
Constrained type = (
```

```
Type: (((),((uv10[faust_min-0,faust_max+0])^uv11)),
```

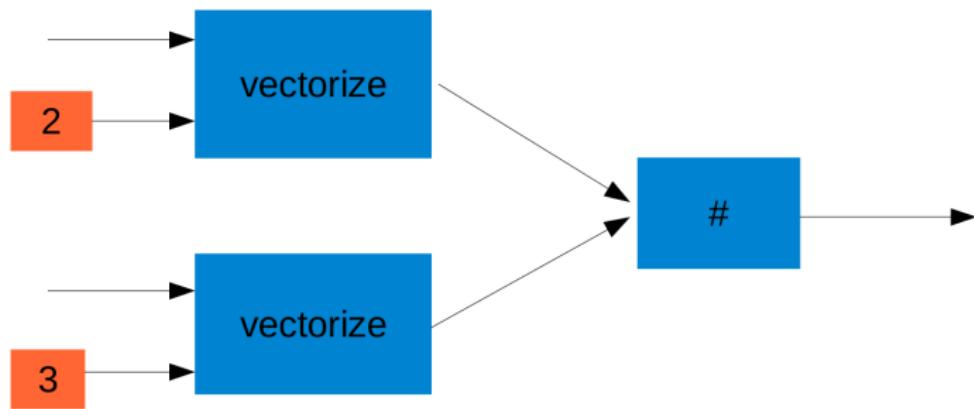
```
Constraint:
```

```
((uv1<=10,uv2>=10,uv11==uv15,uv10==uv12,uv5+uv8>=uv13,uv6+uv9<=uv14,uv11==uv15,  
uv4==uv12,uv5>=uv13,uv6<=uv14,uv3==uv11,Int==uv7,uv1>=uv8,uv2<=uv9),  
(uv1,uv2,uv3,uv4,uv5,uv6,uv7,uv8,uv9,uv10,uv11,uv12,uv13,uv14,uv15)))
```

```
Type = (((),((Int[faust_min-0,faust_max+0])^1))
```

Type checking examples and results

```
process = (_ , 2 : vectorize) , (_ , 3 : vectorize) : # ;
```



Type checking examples and results

```
process = (_,2:vectorize),(_,3:vectorize):# ;
```

Constrained type = (

Type:

```
((uv1[uv2,uv3])^uv4,(uv16[uv17,uv18])^uv19),  
((vector_uv34+uv35(uv31[uv32,uv33]))^uv36)),
```

Constraint:

```
((uv5<=2,uv6>=2,uv4==uv14,uv1==uv8,uv2>=uv9,uv3<=uv10,uv7==uv15,Int==uv11,  
uv5>=uv12,uv6<=uv12,uv20<=3,uv21>=3,uv19==uv29,uv16==uv23,uv17>=uv24,  
uv18<=uv25,uv22==uv30,Int==uv26,uv20>=uv27,uv21<=uv27,uv14/uv12==uv36,  
uv12==uv34,uv8==uv31,uv9>=uv32,uv10<=uv33,uv29/uv27==uv36,  
uv27==uv35,uv23==uv31,uv24>=uv32,uv25<=uv33),  
(uv1,uv2,uv3,uv4,uv5,uv6,uv7,uv8,uv9,uv10,uv11,uv12,uv13,uv14,uv15,uv16,uv17,  
uv18,uv19,uv20,uv21,uv22,uv23,uv24,uv25,uv26,uv27,uv28,uv29,uv30,uv31,uv32,  
uv33,uv34,uv35,uv36)))
```

```
Type = (((Int[0,0])^2,(Int[0,0])^3),((vector_5(Int[0,0])))^1))
```

Conclusion

- Faustine + Faust Type checker = interpreter + type checker for the multirate version of Faust
- Link between the classic typing approach, based on substitutions, and the constraint programming approach
- Future work:
 - ▶ Performance statistics on type checking benchmarks
 - ▶ Constraint solving \implies Constraint programming
 - ▶ Study of different combinations between the typing and constraint programming approaches
 - ▶ Possible case of study: Optimization of the loop case in the Faust syntax
 - ▶ Integration into the C++ compiler of Faust

Selective bibliography

- [1] Jouvelot, P., and Gifford, D. K.
Algebraic Reconstruction of Types and Effects.
In Proceedings of the 1991 ACM Conference on Principles of Programming Languages.
ACM, New-York, 1991.
- [2] Jouvelot, P., and Orlarey, Y.
Dependent vector types for data structuring in multirate Faust.
In Computer Languages, Systems & Structures Journal.
Elsevier, 2011.
- [3] Milner, R.
A Theory for type polymorphism in programming.
In Journal of Computer and Systems Sciences, Vol. 17, pages 348-375.
1978.



A Constraint-Solving Approach to Faust Program Type Checking

Constraint Programming Meets Verification 2014 Workshop

Imré Frotier de la Messelière¹, Pierre Jouvelot¹, Jean-Pierre Talpin²

¹MINES ParisTech, PSL Research University

²INRIA

September 8, 2014