A Constraint-Solving Approach to Faust Program Type Checking

Constraint Programming Meets Verification 2014 Workhsop

Imré Frotier de la Messelière[1], Pierre Jouvelot[1], Jean-Pierre Talpin[2]

[1]MINES ParisTech, PSL Research University

[2]INRIA

September 8, 2014

Contents

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 $\Longrightarrow$ 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 $\Longrightarrow$ (( ) , (int $[-2,2]$))

2 $\Longrightarrow$ (( ) , (int $[0,3]$))

+ $\Longrightarrow$ ((int $[-20,20]$, int $[-20,20]$), (int $[-40,40]$))

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

Faust program type checking

Typing rules : $\Longrightarrow$ Type specification for programmers

$$\frac{\begin{array}{rcl} T(\mathtt{I}) & = & \Lambda l.(z, z') \\ \forall (x, S) \in l & . & l'(x) \in S \end{array}}{T \vdash \mathtt{I} : (z, z')[l'/l]} \qquad \frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1, z'_1) \\ T & \vdash & \mathtt{E}_2 : (z'_1, z'_2) \end{array}}{T \vdash \mathtt{E}_1 : \mathtt{E}_2 : (z_1, z'_2)} \qquad \frac{\begin{array}{rcl} T & \vdash & \mathtt{E} : (z, z') \\ z' & \subset & z'_1 \\ z_1 & \subset & z \end{array}}{T \vdash \mathtt{E} : (z_1, z'_1)}$$

$$\frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1, z'_1) \\ T & \vdash & \mathtt{E}_2 : (z_2, z'_2) \end{array}}{T \vdash \mathtt{E}_1, \mathtt{E}_2 : (z_1 \| z_2, z'_1 \| z'_2)} \qquad \frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1, z') \\ T & \vdash & \mathtt{E}_2 : (z_2, z'_2) \\ z_2 & = & z'[1, |z_2|] \\ z'_2 & = & z_1[1, |z'_2|] \end{array}}{T \vdash \mathtt{E}_1 \sim \mathtt{E}_2 : (z_1[|z'_2| + 1, |z_1|], \widehat{z'})}$$

Faust program type checking

$$\frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1, z_1') \\ T & \vdash & \mathtt{E}_2 : (z_2, z_2') \\ z_1' & \prec & z_2 \end{array}}{T \quad \vdash \quad \mathtt{E}_1 <: \mathtt{E}_2 : (z_1, z_2')}$$

$$\begin{array}{rcl} z_1' \prec z_2 & = & d_1' d_2 \neq 0 \text{ and} \\ & & \mathrm{mod}(d_2, d_1') = 0 \text{ and} \\ & & \|_{1,d_2,d_1'} \lambda i. z_1' = z_2 \end{array}$$

$$\frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1, z_1') \\ T & \vdash & \mathtt{E}_2 : (z_2, z_2') \\ z_1' & \succ & z_2 \end{array}}{T \quad \vdash \quad \mathtt{E}_1 :> \mathtt{E}_2 : (z_1, z_2')}$$

$$\begin{array}{rcl} z_1' \succ z_2 & = & d_1' d_2 \neq 0 \text{ and} \\ & & \mathrm{mod}(d_1', d_2) = 0 \text{ and} \\ & & \displaystyle\sum_{i \in [0, d_1'/d_2 - 1]} z_1[1 + i d_2, (i+1) d_2] = z_2 \end{array}$$

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++ $\Longrightarrow$ Inclusion within the current Faust compiler
- Based on the type checking algorithm:
  - Input: Faust expression
  - Output: type of the Faust expression or “fail”

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

  + : ((int $[a_1,b_1]$, int $[a_2,b_2]$), (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 : + $\Longrightarrow$ + : ((int [-1,1], int [-1,1]), (int [-2,2]))

- Unification variables = variables for constraints
- Different possible instances, based on subtyping:

  1 , 10 : + $\Longrightarrow$ + : ((int [-1,1], int [-10,10]), (int [-11,11]))

Constraint-solving approach

Predicates syntax

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

$e \in \mathsf{E} ::= \ i \mid o_1\ e \mid e\ o_2\ e$

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

$o_1 \in \mathsf{O}_1 ::= \text{ sin } \mid \text{ cos } \mid \text{ ...}$

$o_2 \in \mathsf{O}_2 ::= \ + \mid - \mid \text{ ...}$

$i \in \mathsf{I}$

Constraints syntax

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

where,

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

Constraint-solving approach

Constrained types

- constrained_type ::= (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

  $\Longrightarrow$ Type (Global result of the algorithm)

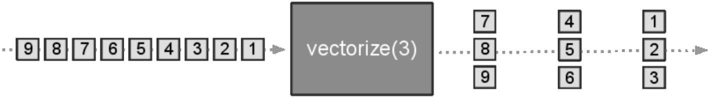
Constraint-solving approach

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

Handling the multirate version of Faust

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



- **serialize**: Input signal at rate $f \implies$ 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 $\Longrightarrow$ (( ) , ($\text{vector}_2(\text{int } [\text{-1,1}])^{22000}$))

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

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

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

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

Handling the multirate version of Faust

Additional environment entries in T:

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

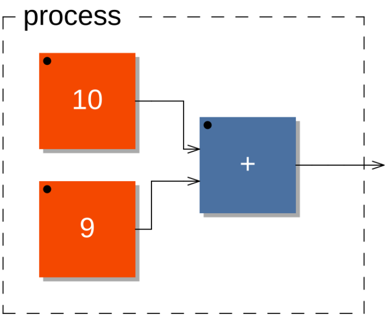
T(serialize) = $(\text{vector}_n(\tau)^f) \longrightarrow (\tau^{f\times n})$

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

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

Type checking examples and results

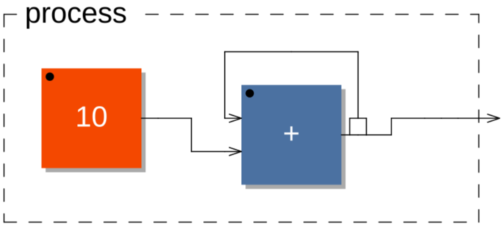
```
process = 10,9:+ ;
```




```
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

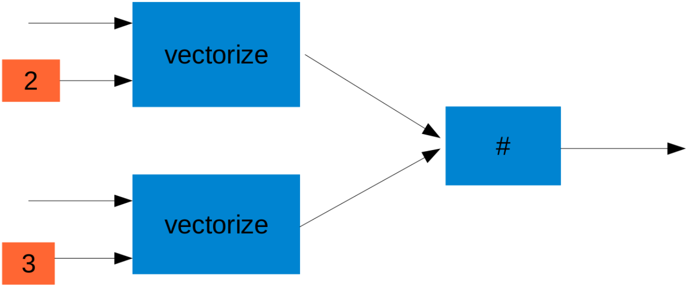
```
process = 10:+~_ ;
```




```
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 $\Longrightarrow$ 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 Workhsop

Imré Frotier de la Messelière[1], Pierre Jouvelot[1], Jean-Pierre Talpin[2]

[1]MINES ParisTech, PSL Research University

[2]INRIA

September 8, 2014