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 \(^\text{[3]}\)
  - the algebraic reconstruction approach of Jouvelot and Gifford \(^\text{[1]}\)
- 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

\[
\text{expression}_{\text{type}} ::= (\text{beam}_{\text{type}},\text{beam}_{\text{type}}) \\
\text{beam}_{\text{type}} ::= \text{signal}_{\text{type}} \text{ list} \\
\text{signal}_{\text{type}} ::= \text{base}_{\text{type}} [x,x] \\
\text{base}_{\text{type}} ::= \text{int} | \text{float} \\
x \in \mathbb{Z}
\]

Type examples

1 \implies ( ( ) , (\text{int} [-2,2])) \\
2 \implies ( ( ) , (\text{int} [0,3])) \\
+ \implies ( (\text{int} [-20,20], \text{int} [-20,20]), (\text{int} [-40,40])) \\
1 , 2 : + \implies ( ( ) , (\text{int} [-40,40]))
Faust program type checking

Typing rules: \[ \Rightarrow \] Type specification for programmers

\[
\begin{align*}
T(\text{I}) & = \forall l. (z, z') \\
\forall (x, S) \in l & . \ l'(x) \in S \\
\frac{T \vdash \text{I} : (z, z')}{T \vdash \text{I} : (z, z')[l'/l]}
\end{align*}
\]

\[
\begin{align*}
T & \vdash E_1 : (z_1, z'_1) \\
T & \vdash E_2 : (z'_1, z'_2) \\
\frac{T \vdash E_1 : E_2 : (z_1, z'_2)}{T \vdash E_1 : E_2 : (z_1, z'_1)}
\end{align*}
\]

\[
\begin{align*}
T & \vdash E_1 : (z_1, z'_1) \\
T & \vdash E_2 : (z_2, z'_2) \\
\frac{T \vdash E_1 \sim E_2 : (z_1 [z_2] + 1, z_1 [1], z')}{T \vdash E_1 \sim E_2 : (z_1 [1], z_2, z'_1, z'_2)}
\end{align*}
\]
Faust program type checking

\[
\begin{align*}
T \vdash & E_1 : (z_1, z_1') \\
T \vdash & E_2 : (z_2, z_2') \\
\quad z_1' \prec z_2 & = d_1' d_2 \neq 0 \text{ and } \\
& \quad \mod(d_2, d_1') = 0 \text{ and } \\
& \quad \|_{1, d_2, d_1'} \lambda i. z_1' = z_2 \\
\hline
T \vdash & E_1 \ll E_2 : (z_1, z_2') \\
\end{align*}
\]

\[
\begin{align*}
T \vdash & E_1 : (z_1, z_1') \\
T \vdash & E_2 : (z_2, z_2') \\
\quad (z_1' \preceq z_2 & = d_1' d_2 \neq 0 \text{ and } \\
& \quad \mod(d_1', d_2) = 0 \text{ and } \\
& \quad \sum_{i\in[0,d_1'/d_2-1]} z_1[1 + id_2, (i + 1)d_2] = z_2 \\
\hline
T \vdash & E_1 :> E_2 : (z_1, z_2') \\
\end{align*}
\]
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)
  
  \[
  1, 1 : + \quad \Rightarrow \quad + : ( (\text{int } [-1,1], \text{int } [-1,1]), (\text{int } [-2,2]) )
  \]

- Unification variables = variables for constraints

- Different possible instances, based on subtyping:
  
  \[
  1, 10 : + \quad \Rightarrow \quad + : ( (\text{int } [-1,1], \text{int } [-10,10]), (\text{int } [-11,11]) )
  \]
**Constraint-solving approach**

**Predicates syntax**

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

\[
e \in E ::= i \mid o_1 \ e \mid e \ o_2 \ e
\]

\[
b \in B ::= = \mid < \mid \leq \mid > \mid \geq
\]

\[
o_1 \in O_1 ::= \sin \mid \cos \mid ...
\]

\[
o_2 \in O_2 ::= + \mid - \mid ...
\]

\[i \in I\]

**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

- 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
  \[ \implies \text{Type (Global result of the algorithm)} \]
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 $f \implies$ Output signal at rate $f/n$

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

- `[ ]`: element access

- `#`: concatenation
Type syntax for a Faust multirate expression

expression_type ::= (beam_type, beam_type)
beam_type ::= signal_type list
signal_type ::= faust_type \text{rate}
faust_type ::= base_type [x,x] \mid \text{vector}_n(\text{faust_type})
base_type ::= \text{int} \mid \text{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 \[\implies ( ( ) , ( \text{vector}_2(\text{int }[-1,1])^{22000} ) )\]

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

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

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

1, 2 : vectorize, 3 : vectorize \[\implies ( ( ) , ( \text{vector}_3(\text{vector}_2(\text{int }[-1,1]))^{22000} ) )\]
Handling the multirate version of Faust

Additional environment entries in $T$:

$T($vectorize$) = (\tau^f, \text{int}[n,n]^f) \rightarrow (\text{vector}_n(\tau)^{f/n})$

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

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

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

Algebraic Reconstruction of Types and Effects.

Dependent vector types for data structuring in multirate Faust.
Elsevier, 2011.

A Theory for type polymorphism in programming.
1978.
A Constraint-Solving Approach to Faust Program Type Checking

Constraint Programming Meets Verification 2014 Workshop

Imré Frotier de la Messelière\textsuperscript{1}, Pierre Jouvelot\textsuperscript{1}, Jean-Pierre Talpin\textsuperscript{2}

\textsuperscript{1}MINES ParisTech, PSL Research University

\textsuperscript{2}INRIA

September 8, 2014