A Constraint-Solving Approach to Faust Program Type Checking

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

[1]MINES ParisTech, PSL Research University

[2]INRIA

October 13, 2014

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

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

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

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

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

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

$i \in \mathsf{I}$

Constraints syntax

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

where,

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

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)

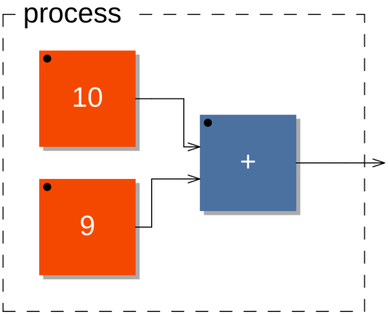
Constraint-solving approach

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

Type checking examples and results

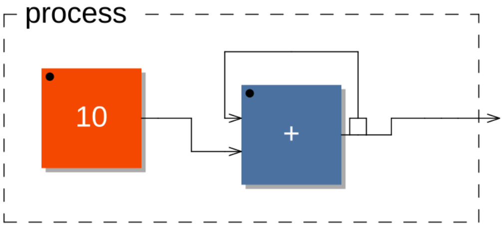
```
process = 10,9:+ ;
```




```
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

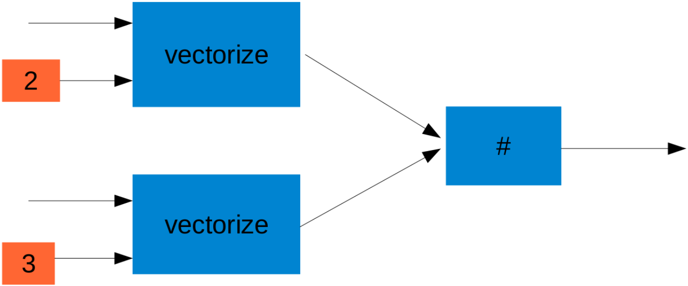
```
process = 10:+~_ ;
```




```
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

A Constraint-Solving Approach to Faust Program Type Checking

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

[1]MINES ParisTech, PSL Research University

[2]INRIA

October 13, 2014