Formal analysis and implementation of the Faust programming language

Report of first PhD year

Imré Frotier de la Messelière[1]

PhD advisor: Pierre Jouvelot[1]

PhD co-advisor: Jean-Pierre Talpin[2]

[1]MINES ParisTech, CRI
[2]INRIA, IRISA

May 26, 2014

Abstract

This report presents the current overall status of my PhD thesis work. My goal is to optimize the compiler of the Faust programming language [16], a domain specific language dedicated to audio signal processing. In order to achieve this objective, I have worked on a type inference algorithm, taking inspiration from the algorithm W of Hindley-Milner [5] and on the algebraic reconstruction algorithm of Jouvelot and Gifford [9]. My ultimate goal is to provide Faust with a formally proven static typing system, thus making it more reliable and efficient. The type inference algorithm presented here is organized in two main parts: a classic type inference algorithm, coupled with the generation of constraints, and a solver to determine if the resulting constraints system is decidable and to provide a mapping yielding the type of the input expression.

Including an abstract interpretation [4] aspect to this type system is part of the current tracks of studies for this algorithm. Moreover, an implementation in OCaml is currently being developped. This is a first prototype that will then be rewritten in C++ in order to be inserted inside the actual Faust compiler, so that it may be made available along with the official Faust release. Synchronicity is an additional area of study as well. In addition to these, I provide a previsional planning for the remaining time to complete my PhD thesis. I also present the skills acquired through doctoral courses, as well as my English language proficiency. Finally, a principal bibliography summarizes the related work quoted throughout this document.

Keywords : abstract interpretation, constraint, dependent type, Faust, Hindley-Milner, signal processing, solver, static typing, synchronous language, type checking, type inference, type system.

Contents

1 Introduction **3**

2 PhD thesis overview **4**

2.1 Subject and context . . . 4

2.1.1 “Formal analysis and implementation of the Faust programming language” . . . 4

2.1.2 FEEVER : “Faust Environment Everyware” . . . 4

2.2 Presentation of Faust . . . 4

2.2.1 Overview and syntax . . . 4

2.2.2 Static semantics . . . 6

2.3 Stakes and challenges . . . 6

2.3.1 Interest and stakes for Faust . . . 6

2.3.2 General scientific contributions . . . 6

2.4 Related work . . . 7

2.4.1 Main references with links to type systems . . . 7

2.4.2 References on synchronicity . . . 7

3 Type checking algorithm for the Faust programming language **8**

3.1 Types and specifications . . . 8

3.1.1 Types and notations . . . 8

3.1.2 Predicate specification . . . 8

3.1.3 Constraint specification . . . 9

3.1.4 Constrained type specification . . . 10

3.2 Type checking . . . 10

3.2.1 Main algorithm . . . 10

3.2.2 Constrained type computation . . . 10

3.2.3 Lower level functions . . . 12

3.3 Correctness theorems . . . 13

3.3.1 Soundness . . . 13

3.3.2 Completeness . . . 13

4 Current work and study perspectives **14**

4.1 Completion of the current algorithm . . . 14

4.1.1 Solver . . . 14

4.1.2 Proofs of the theorems . . . 14

4.2 New elements for the current algorithm . . . 15

4.2.1 Multirate Faust . . . 15

4.2.2 Completion of the implementation . . . 15

4.3 Further improvements . . . 15

4.3.1 Use of abstract interpretation . . . 15

4.3.2 Integration into the Faust compiler . . . 16

5 Additional prospective approaches **16**

5.1 Polychronicity . . . 16

5.2 Additional synchronicity studies . . . 16

5.3 Other tracks of study . . . 16

6 Doctoral courses and other skills **16**

6.1 Scientific courses . . . 16

6.2 Professionalizing courses . . . 17

6.3 English language proficiency . . . 17

7 Previsional planning **17**

8 Conclusion **17**

9 Bibliography **18**

1 Introduction

This report presents the current overall status of my PhD thesis work for this first year. My goal is to optimize the compiler of the Faust programming language [16], a domain specific language dedicated to signal processing. In order to achieve this objective, I have worked on a type inference algorithm, taking inspiration from the usual algorithm W of Hindley-Milner [5] and on the algebraic reconstruction algorithm of Jouvelot and Gifford [9]. My ultimate goal is to provide Faust with a formally proven static typing system, thus making it more reliable and efficient, as static typing can find type errors reliably at compile time and usually results in compiled code that executes more quickly. Moreover, it can enable the compiler to produce optimized machine code. This whole work is part of the ANR project FEEVER, which promotes the optimization and ubiquity of Faust and the applications relying on it. This document presents the type inference algorithm I am currently designing for Faust in order to optimize its compiler. The algorithm presented here is organized in two main parts: a classic type inference algorithm, coupled with the generation of constraints, and a solver to determine if the resulting constraints system is decidable and to provide a mapping of unification variables toward type values.

Here, we shall focus more in depth on the constraints generation part of the algorithm. It outputs a constrained type, which is a pair composed of a type and a constraint, which may contain multiple predicates. A constrained type can be considered as a member of the larger class of dependent types [25]. Due to the association of a constraint to a type, this notion also has ties to refinement types [8] and liquid types [17]. However, our approach is different from the one of these existing types because our Faust types contain intervals, which would be considered as constraints in usual dependent types. Among already existing concepts, there are a few different ones which have their own notion of constrained types, but these are not the ones we are interested in. Instead, we introduce our own notion of constrained type as well, which relies on constraints based on a different specification.

The second part of the algorithm is a solver which determines if the resulting constraints system is decidable and provides a mapping of unification variables toward type values. In order to design this solver, I am basing my work on existing projects such as Z3 [15] from Microsoft. My objective is to design a lighter solver using only theories that are required by the specification of my constraints system. Currently, I am working on a translation of the instances of this system towards the smt-lib language [2] so that I may provide the output constraints of the first part of my algorithm as input to Z3 and other existing solvers. This way, I may use this existing work as part of a prototype, knowing that the final implementation of my algorithm will nonetheless have to be autonomous in order to be integrated in the compiler of Faust.

Including an abstract interpretation [4] aspect to this type system is part of the current tracks of studies for this algorithm. This would enable me to optimize the handling of the recursive loop case of the Faust syntax during the execution of my type inference algorithm. My goal is to use a better approximation through abtract interpretation, first basing myself on works like [23] and [21], which provide a first view on an abstract semantics for type systems, with [23] focusing on the case of static type systems. I am currently studying three different approaches. The first two approaches take place in the first part of the algorithm, inserting this construction either in the typing rules themselves or outside them. The third approach is to insert an abstract interpretation threory in the theories module of the solver.

Moreover, an implementation in OCaml, using the ground work laid by the Faustine interpreter [1], is currently being developped. This is a first prototype that will then be translated into C++ in order to be inserted inside the actual Faust compiler, so that it may be made available along with the official Faust release. Based on this type checking algorithm, and using the classes infrastructure that is introduced throughout this document, it shall take a Faust expression as input and output its type if it runs successfully.

Considering all of the above, this PhD thesis deals with the following issues. It introduces a static typing system for Faust thanks to type inference. The handling of undecidability and of intervals is taken care of by constraints and abstract interpretation respectively. All in all, I work on providing a synchronous system combining type inference, constraints and abstract interpretation. Thus, my presentation is organized as follows. First, I provide an overview of my PhD thesis work in Section 2, with Subsection 2.1 dedicated to the thesis summary and its context, Subsection 2.2 being a reminder of the basic notions of Faust, Subsection 2.3 focusing on the most interesting challenges provided by this thesis, and Subsection 2.4 giving more insight on the existing publications related to my work.

The type checking algorithm of Faust is introduced in Section 3. Subsection 3.1 presents the types and notations used in the algorithm, along with the specifications of predicates and constraints, and an introduction to constrained types. These will serve as a work basis for both the theoretical design and implemention of the type checking algorithm. After these introductory preparations, the complete algorithm is presented in Subsection 3.2. Subsection 3.3 contains the conjectured theorems of soundness and completeness for the algorithm. Section 4 presents my current work and study perspectives. In addition to these, I also provide perspectives for other approaches in Section 5. After this, a presentation of my doctoral courses and English language skills will be detailed in Section 6. A previsional planning is then made available in Section 7. Finally, Section 8 concludes and the main bibliography of my work is made available in Section 9.

2 PhD thesis overview

2.1 Subject and context

2.1.1 “Formal analysis and implementation of the Faust programming language”

The Faust programming language (Functional Audio Stream), designed at GRAME, a national musical creation center located in Lyon, focuses on the definition of synchronous digital audio signal processes. The Faust programming paradigm is strictly functional. This language, used all over the world, can be applied in various domains, from advanced audio filter design to innovative contemporary music installations. A first formal analysis of the core of Faust has been developed at CRI, as part of the ANR Astrée project. This analysis has led to the definition of the precise semantics of the language core instruction set and a first attempt at characterizing the language key semantic properties (typing, synchronicity). The goal of this PhD research is to:

- extend the existing formal definitions of the language core to the whole language,
- extend and generalize the existing theorems regarding Faust mathematical properties,
- implement within the Faust compiler the analyses developed above in order to provide even more performant implementations (sequential, parallel) of Faust programs.

2.1.2 FEEVER : “Faust Environment Everyware”

My PhD work takes place as part of the ANR project FEEVER. Started on October 2013, it is structured around four partners: ARMINES (project lead), CIEREC, GRAME, and the IRISA center of INRIA. Nowadays, music and more generally audio processing is a life-enhancing practice that impacts everyone everywhere in evermore personalized manners. Yet, the Net has been slow to offer users and developers the kind of advanced technologies that would make similar listening enrichment standard, seamless and easily customizable online. FEEVER intends to make such a vision a reality. Yet, scientific and technical challenges abound, while the technological solutions need to be:

- portable, to allow program-once, deploy-everywhere economic advantages;
- easily programmable, to narrow the gap between specifications and implementation;
- able to deal with multiple platforms, for seamless integration within the user’s listening environments;
- efficient both in terms of computing time, since audio processing is a highly compute-intensive activity, and energy, if only to permit mobile applications;
- secure, since audio processing activity performed on the client side must not jeopardize the user system.

It seems the Faust programming ecosystem introduced at GRAME 10 years ago is the proper starting point towards the global solution the audio world is waiting for. FEEVER intends to make this happen via these main tasks:

- “Task 1 : Models”, which focuses on formal issues at the language level,
- “Task 2 : Compiler”, which intends to provide an industrial-strength, efficient, multi-rate, multi-platform, portable, easily integrable Faust compiler,
- “Task 3 : Ubiquity”, which puts FEEVER on the global scene, looking at ways to make Faust-enabled solutions available everywhere, be it as a Web service or integrated within a smartphone web browser.
- “Task 4 : Education”, which strives to make FEEVER technologies even more relevant by looking at the new usages they open up for audio processing and, more generally, digital signal processing teaching.

2.2 Presentation of Faust

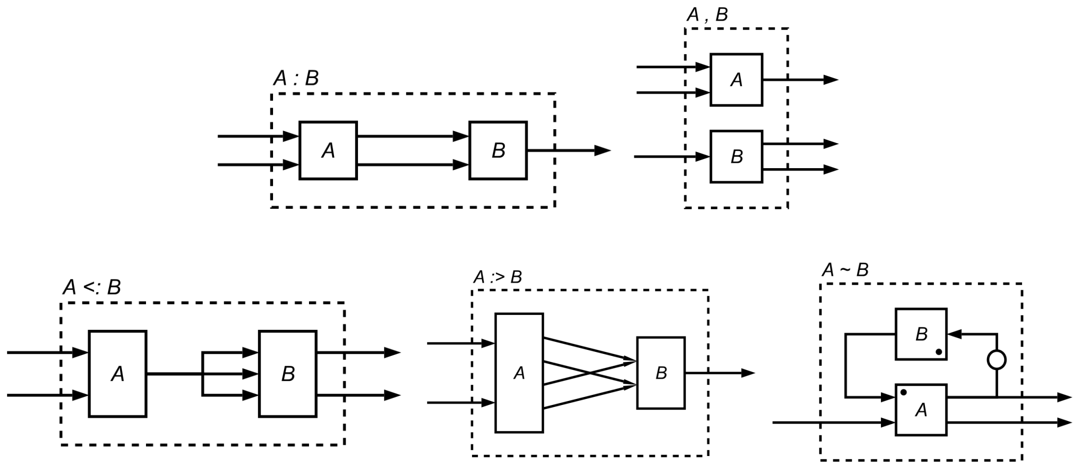
2.2.1 Overview and syntax

A Faust program describes a signal processor, which is a function that gets input signals and produces output signals. These signals are themselves functions of time to values. A distinctive characteristic of Faust is to be fully compiled. The Faust compiler translates digital signal processing specifications into C++ code. The generated code is self-contained and doesn’t depend on any library or runtime.

The syntax for the core of Faust is the following, where e is a Faust expression and i is an identifier:

$$e ::= i \mid e_1 : e_2 \mid e_1, e_2 \mid e_1 <: e_2 \mid e_1 :> e_2 \mid e_1 \sim e_2$$

Case $e_1 : e_2$ corresponds to two connected expressions while case e_1, e_2 corresponds to two parallel expressions. Case $e_1 <: e_2$ stands for the split of the outputs of a Faust expression to the more numerous inputs of another Faust expression. Case $e_1 :> e_2$ is the merging of the outputs of a Faust expression into the fewer inputs of another Faust expression. Finally, case $e_1 \sim e_2$ corresponds to a loop between two Faust expressions. All these expressions correspond to the following blocks diagram elements, where A and B stand for Faust expressions. These figures are provided by [7].

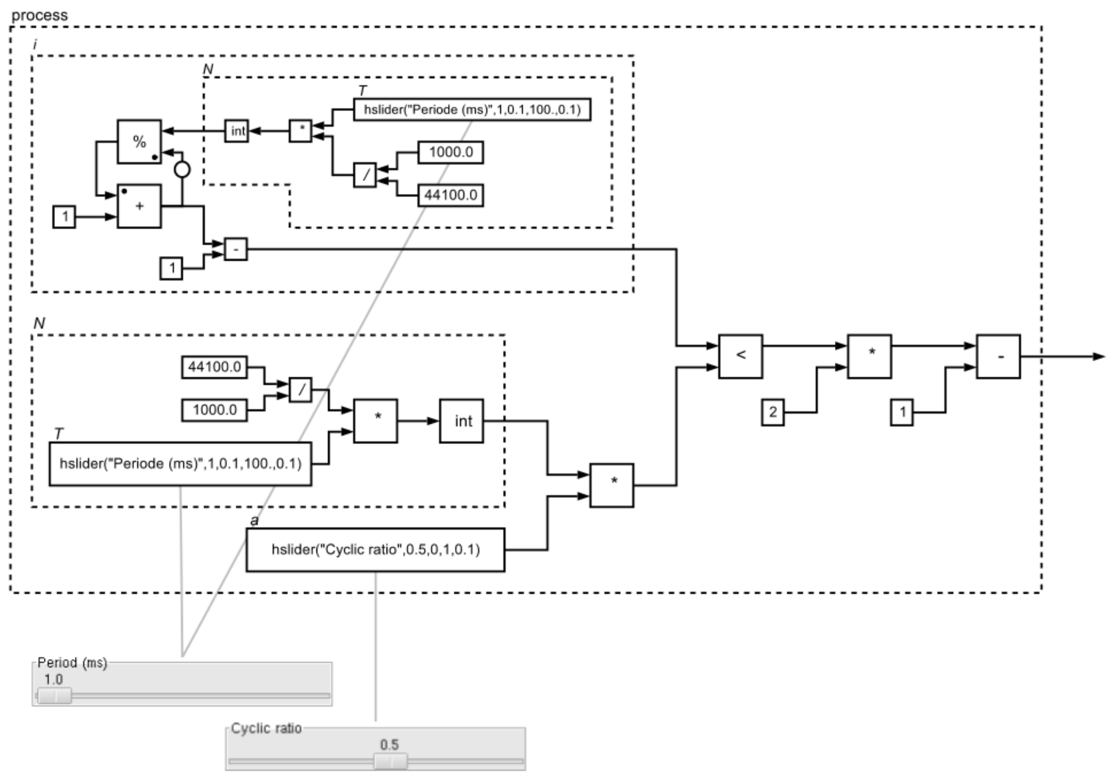


Next is an example of Faust program. The code and the corresponding block diagram are presented below. These figures are provided by [7].

```
//---------------------------------
//     A square wave oscillator
//---------------------------------

T = hslider("Period",1,0.1,100.,0.1); // Period (ms)
N = 44100./1000.*T:int; // The period in samples
a = hslider("Cyclic ratio",0.5,0,1,0.1); // Cyclic ratio
i = +(1)~%(N):-(1); // 0,1,2...,n

process = i,N*a : < : *(2) : -(1) ;
```


process
i
N
T
hslider("Periode (ms)",1,0.1,100.,0.1)
int
*
%
1000.0
/
44100.0
+
1
1
-
N
44100.0
1000.0
/
*
int
T
hslider("Periode (ms)",1,0.1,100.,0.1)
a
hslider("Cyclic ratio",0.5,0,1,0.1)
*
<
2
*
1
-
Period (ms)
1.0
Cyclic ratio
0.5

2.2.2 Static semantics

The type of a Faust expression is a pair composed of a base type and an interval. The interval indicates the possible range of an instance of this type. For example, 1 has type (int,$[0,1]$) in Faust. However, Faust expressions may have several possible type assignations. Here, the Faust expression 1 may also have type (int,$[-2,2]$), (float,$[1,1]$) or even (float,$[-\infty,+\infty]$).

The static semantics of Faust stands as follows. It provides the types of the expressions of the syntax of Faust, as well as the conditions for these expressions to exist. The following figures are provided by [11].

The rule for identifiers instantiates type variables with nondescript types. The sequence rule makes sure the connecting types match. The parallel rule simply appends the types of the expressions.

$$\frac{\begin{array}{rcl} T(\mathtt{I}) & = & \Lambda l.(z,z') \\ \forall (x,S)\in l & . & l'(x)\in S \end{array}}{T \;\vdash\; \mathtt{I} : (z,z')[l'/l]} \qquad \frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1,z_1') \\ T & \vdash & \mathtt{E}_2 : (z_1',z_2') \end{array}}{T \;\vdash\; \mathtt{E}_1 : \mathtt{E}_2 : (z_1,z_2')} \qquad \frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1,z_1') \\ T & \vdash & \mathtt{E}_2 : (z_2,z_2') \end{array}}{T \;\vdash\; \mathtt{E}_1,\mathtt{E}_2 : (z_1\|z_2, z_1'\|z_2')}$$

The splitting rule makes sure the connections between the two expressions are complete and describes how the signals are dispatched.

$$\frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1,z_1') \\ T & \vdash & \mathtt{E}_2 : (z_2,z_2') \\ z_1' & \prec & z_2 \end{array}}{T \;\vdash\; \mathtt{E}_1 \mathrel{<:} \mathtt{E}_2 : (z_1,z_2')} \qquad \begin{array}{rcl} z_1' \prec z_2 & = & d_1'd_2 \neq 0 \text{ and} \\ & & \mathrm{mod}(d_2,d_1') = 0 \text{ and} \\ & & \|_{1,d_2,d_1'}\lambda i.z_1' = z_2 \end{array}$$

The merging rule relies on the same principle.

$$\frac{\begin{array}{rcl} T & \vdash & \mathtt{E}_1 : (z_1,z_1') \\ T & \vdash & \mathtt{E}_2 : (z_2,z_2') \\ z_1' & \succ & z_2 \end{array}}{T \;\vdash\; \mathtt{E}_1 \mathrel{:>} \mathtt{E}_2 : (z_1,z_2')} \qquad \begin{array}{rcl} z_1' \succ z_2 & = & d_1'd_2 \neq 0 \text{ and} \\ & & \mathrm{mod}(d_1',d_2) = 0 \text{ and} \\ & & \displaystyle\sum_{i\in[0,d_1'/d_2-1]} z_1[1+id_2,(i+1)d_2] = z_2 \end{array}$$

The loop rule handles the recursive connections and widens the interval of the output type to infinity. The final rule stands for subtyping.

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

2.3 Stakes and challenges

2.3.1 Interest and stakes for Faust

My PhD work has an impact on Faust, as it strives to optimize its compiler. The development of Faust is of noticeable interest since it is used in various domains all over the world. Introducing a static typing promotes safety for Faust. It delivers approximations of the cases of undecidability during type checking. Moreover, my system shall handle intervals in a more precise fashion than is currently done with the current approximation to infinity during the unfolding of the loop cases.

A most important track of work is the extension of Faust to its multirate version, which can handle signals with different frequencies, as well as vector structures. It does not exist at all within the official Faust compiler for now. Thus, integrating this extension would be a leap forward for Faust.

In addition to introducing this static typing system and the multirate version of Faust, another point of interest is to prove properties of Faust programs that are currently conjectured. Handling the macro-expansion phenomenon of the implementation of the multirate version of Faust is also a stake that is worthy of notice. The multirate version of Faust handles sets of signals that may have different frequencies.

2.3.2 General scientific contributions

Currently, my main contribution is the design of a type inference system, relying on a type checking algorithm. As part of this algorithm, abstract interpretation shall be used in the handling of intervalls, which are currently simplified to infinity by the widening function. In the end, my goal is to provide not only a synchronous system combining type inference, constraints and abstract interpretation, but also an extension to multirate as well as a new type system for an unusual language.

2.4 Related work

2.4.1 Main references with links to type systems

The first set of references is related to the Faust programming language and encompasses very useful information about the current status of Faust. Mainly, it presents the base notions of Faust, along with its semantics and its multirate extension.

- Presentation of Faust : Yann Orlarey, Dominique Fober, Stephane Letz [16].
- Multirate version of Faust : Pierre Jouvelot, Yann Orlarey [11] [12].
- Faustine, a Faust interpreter and prototype for multirate Faust : Karim Barkati, Haisheng Wang, Pierre Jouvelot [1].

The second set of references deals with multiple general aspects of type systems, especially on type inference, dependent types and constraints. Dependent types are types containing information about the value of their respective instances.

- Type inference and Hindley-Milner algorithm : Luis Damas, Robin Milner [5].
- Dependent types : Hongwei Xi [25].
  - Refinment types : Andrew D. Gordon, Cédric Fournet [8].
  - Liquid types : Patrick M. Rondon, Ming Kawaguchi, Ranjit Jhala [17].
- Constraints : Olivier Tardieu, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat [20], Kenneth L. McMillan, Andrey Rybalchenko [14].

Solvers have also been a strong focus point for my PhD work, since they are at the heart of the second part of my algorithm. I mainly studied the architectures of several existing solvers and the format of their inputs. They rely on the notion of satisfiability modulo theories (SMT), in addition to the usual notion of satisfiability.

- Z3 : Leonardo de Moura, Nikolaj Bjorner [15].
- Gecode : Christian Schulte, Mikael Lagerkvist, Guido Tack. [18],
- veriT : Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, Pascal Fontaine.[3],
- Redlog : Andreas Dolzmann, Thomas Sturm. [6],
- smt-lib language : Clark Barrett, Aaron Stump, Cesare Tinelli. [2].

Abstract interpretation is going to be an even stronger point of focus as my work on the approximation of intervals goes on. Right now, I have studied its most usual aspects, with a focus on abstract semantics for static type systems.

- Original presentation of abstract interpretation: Patrick Cousot, Radhia Cousot [4].
- Type and effect system by abstract interpretation : Jérôme Vouillon, Pierre Jouvelot [23].
- Refinments of abstract types : Niki Vazou, Patrick M. Rondon, Ranjit Jhala [21].

2.4.2 References on synchronicity

In parallel to my work on type systems, I strive to pinpoint the aspects of the synchronous world that may help enhance the functionalities of Faust. I have been able to isolate several promising leads, which may have a positive impact on my thesis in the long run. Up to now, my main area of study on synchronicity has been polychronicity and the use of multiple clocks.

- Polychrony : Paul Le Guernic, Jean-Pierre Talpin, Jean-Christophe Le Lann [13].
- Use of the Polychrony framework : Bin Xue, Sandeep K. Shukla. [26]

Another area of study has been sequential constructivity, including its ability to assign multiple values to a variable during the same time unit, which may prove useful when dealing with the multirate version of Faust: Jean-Pierre Talpin, Jens Brandt, Mike Gemünde, Klaus Schneider, Sandeep Shukla [19].

“End-to-end latency” has also been studied, especially because it provides constraints on synchronous systems : Rémy Wyss, Frédéric Boniol, Claire Pagetti, Julien Forget [24].

Finally, I have also studied synchronous concurrency: Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Bjorn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O’Brien [22].

3 Type checking algorithm for the Faust programming language

3.1 Types and specifications

3.1.1 Types and notations

These are the definitions and notations of the types of the mathematical objects that are handled by the algorithm, whose name is type_checking. Letters in italics a represent an instance of a type. Letters in upper case A represent the set of all instances of a type. Greek letters α represent placeholders and buffers. Types whose naming conventions are not introduced here are not directly used in the parts of the algorithm presented in this document. Instead, they intervene in lower level functions.

boolean operator : b : instance of boolean operator. B : the set of boolean operators.

constraint : Pair containing a set of predicates and a set of identifiers. c : instance of constraint. C : the set of constraints.

constrained type : Pair containing a type and a constraint. This is the output of the first part of the algorithm. k : instance of constrained type. K : the set of constrained types.

environment : Mapping of variables on type scheme values, which is an input of the algorithm and is not modified during the execution of the algorithm. r : instance of environment. R : the set of environments.

expression : Faust expression, which is an input of the algorithm. The type of a Faust expression corresponds to the pair containning the type of its input beam and the type of its output beam. The operations on expression types are the same as the operations on expressions. e : instance of expression. E : the set of expressions.

identifier : Variable name, which enables me to avoid duplicates. Examples of identifiers are instance names and symbols representing operators. A particular case of identifier is the case of unification variables, that are implementing variables on type schemes. i : instance of identifier. I: the set of identifiers.

mapping : Mapping of identifiers to type values. m : instance of mapping. M : the set of mappings.

numerical operator : o : instance of numerical operator. O : the set of numerical operators.

predicate : Comparison between basic types or between interval boundaries. p : instance of predicate. P : the set of predicates.

signal : Member of a beam; its type is a Faust type, noted type for our algorithm.

type : Faust type. t : instance of type. T : the set of types.

type of beam : Type of a list of signals. These lists of signals are used as input and as output for a whole Faust expression as well as for the blocks it contains. z : instance of a type of beam. Z : the set of types of beams.

type scheme : Type in which the type variables have not yet been instantiated with unification variables or type values. In my design, a type scheme is represented as a pair whose first element is a type and whose second element is a list of identifiers, which stand for the type scheme variables.

3.1.2 Predicate specification

A predicate is a boolean-valued function f : $x \longrightarrow$ true, false, called the predicate on x. In our case, a predicate takes as argument a comparison x between basic types or between interval boundaries. Thus, it represents comparisons between types and also between numerical values, be they integers or floats.

Predicates can be regrouped in sets of predicates, which are conjunctions of predicates. Instead of using multiple constraints with only one predicate, I use a single constraint with multiple predicates. As the notion of set of predicates is conceptually equivalent to a predicate formed by conjunction of other predicates, I shall often use both of these designations for fluidity of speech.

Given an environment r, a predicate (or a set of predicates) can be satisfiable, unsatisfiable, or its satisfiability cannot be determined. For example, with an empty environment, $x \leq 10$ is satisfiable because it evaluates to true for x = 9. Also, a set containing two contradictory predicates is unsatisfiable.

Predicates are implemented according to the following syntax.

$p \in$ P ::= true | $e\ b\ e$

$e \in$ E ::= i | $e\ o\ e$

$o \in$ O ::= + | $\times$

$b \in$ B ::= = | < | >

3.1.3 Constraint specification

A constraint is a pair containing a set of predicates and a set of identifiers. As in the case of predicates, the notion of set of constraints is close to a constraint formed by union of other constraints. So I shall often use both of these designations for fluidity of speech. Given an environment n, a constraint can be satisfiable, unsatisfiable, or its satisfiability cannot be determined. The satisfiability of a constraint is equivalent to the satisfiability of its set of predicates. If the satisfiability of the constraint cannot be determined, the type checking algorithm may not necessarily stop.

I only consider as constraint material the predicates that cannot be evaluated immediately and that actually provide information leading to the mapping of unification variables toward type values.

Here, I present the operations specifically designed for these constraints.

Let c , $c' \in$ C
Let $\rho \in \mathcal{P}$(P) and $\iota \in \mathcal{P}$(I) so that c = (ρ,ι)
Let $\rho' \in \mathcal{P}$(P) and $\iota' \in \mathcal{P}$(I) so that c' = (ρ',ι')

newC : constructor

newC = ($\varnothing$,$\varnothing$)

$c\#1$: set of predicates of constraint c

$c\#1$ = ρ

$c\#2$: set of identifiers of constraint c

$c\#2$ = ι

$c \cup c'$: union of 2 constraints

$c \cup c'$ = ($\rho \cup \rho'$, $\iota \cup \iota'$)

Using the notations from above, we can also write this definition as :

$c \cup c'$ = ($c\#1 \cup c'\#1$, $c\#2 \cup c'\#2$)

The use of $\cup$ ensures that there is no syntactically duplicate predicate in the resulting set of predicates.

Using all of the above, I can now define the syntax of the constraints by adding these statements to the syntax of predicates:

C = $\mathcal{P}$(P) $\times$ $\mathcal{P}$(I)

$c \in$ C ::= $c \cup c$ | (ρ,ι)

ρ ::= $p \cup p$

ι ::= i | $i \cup \iota$

ρ and ι are placeholders used for the writing purposes of this syntax.

3.1.4 Constrained type specification

An instance of a constrained type is a pair whose first element is a type and whose second element is a constraint. The first part of our algorithm, i.e. the usual type inference part, returns a constrained type as output. This enables me to express the fact that I have a type t still containing unification variables, which will be mapped to actual type values by the solver part of our algorithm. The solver does not need t as an input, but only the constraint c. t will only be used again after the execution of the solver, so that the the mapping returned by the solver may be applied to it and thus yield the type that will be the output of the whole algorithm.

Thus, a constrained type has the following designation:

Let $k \in \mathrm{K} = \mathrm{T} \times \mathrm{C}$

$k = (t,c)$ where $t \in \mathrm{T}$ and $c \in \mathrm{C}$

Due to the association of a constraint to a type, this notion has ties to dependent types [25], refinement types [8] and liquid types [17]. However, my approach is different from the one of these existing types because Faust types contain intervals, which would be considered as constraints in usual dependent types. Among already existing concepts, there are a few different ones which have their own separate notions of constrained types, but these are not the ones I am interested in because they use a different specification for their constraints. From the point of view of usual dependent types, I am using two different levels of constraints: the interval and the constraint itself. In my case, I do not directly classify the interval as a constraint but instead extract and use the related data during the computation of constraints.

3.2 Type checking

3.2.1 Main algorithm

The principle of this type inference algorithm is based on [5] and [9]. The notations that are used here are the ones presented in section 3.1. The syntax of Faust and the rules leading to the constraints and structural conditions presented below can be found in [11] and [12]. The name of this algorithm is type_checking. It is organized in two main parts: constrained_type and solve. Thus, type_checking is the function with the highest level of abstraction in the algorithm. It takes as input an environment, a Faust expression and a preliminary constraint, which enables me to provide predefined conditions.

First, in constrained_type, the input Faust expression is explored like in the usual type inference algorithm [5], while constraints are stored at the same time. These constraints are due to the connection of all the blocks of the expression, following the rules described in [11] and [12]. At the end of this first part, we have obtained a type which still contains unification variables that need to be assigned to type values, along with a set of constraints. In our design concept, this set is represented as one constraint which contains a set with all the corresponding predicates. What we now need is to find a mapping of the unification variables to type values. Therefore, we call the second part of our program (a.k.a. solve), which determines whether our constraint system is satisfiable or not. If so, it will return a mapping of the unification variables to type values. Once this is done, the algorithm returns the type of the Faust expression, which is the application of this mapping to the type which we got as an output of constrained_type. If the constraint system is not satisfiable, then the algorithm returns fail.

All of this yields the following outline, which corresponds to the highest level of abstraction of the algorithm.

```
type_checking : R $\longrightarrow$ E $\longrightarrow$ C $\longrightarrow$ T + fail

type_checking $r$ $e$ $c$ =
**let** $(t, c')$ = constrained_type $(r,c)$ $e$
**let** m = solve $c'$
m $t$
```

3.2.2 Constrained type computation

As in [5] and [9] , I perform the basis of a type inference algorithm by reasoning by induction on the input Faust expression e. In each case, we will generate the corresponding constrained type, thus yielding a type containing unification variables to be mapped by the solver part of the main algorihm along with the constraint that will be fed to this solver in order to determine the resulting mapping. These constraints will allow us to keep track of the unification variables to map, and to check whether the type equality conditions that could not be evaluated immediately are satisfiable or not.

- In case $e = i$, I instantiate the type scheme variables with unification variables. In order to do so, we generate as many new unification variables as there are uninstantied type scheme variables.
  Then we substitute these new unification variables to the type scheme variables in the type t. Here, the type scheme variables are represented as the list of identifiers l, while the resulting unification variables are represented as the list of identifiers l'.
  The related constraint is the union of the input constraint and of the constraint containing the empty predicate and the list of identifiers of the new unification variables. Thus, these new unification variables can be tracked in the resulting type t' while applying the solver part of the main algorithm.
  The output of this case is the constrained type consisting of the resulting type t' and the resulting constraint.
  The structural condition that is checked immediately is that i be in the definition domain of the environment n, which is an input of constrained_type, as well as an input of the main algorithm.
  This case corresponds to the lowest level of the expression, like a leaf in a tree.

- In case $e = e_1 : e_2$, I first recursively call constrained_type on e_1 and e_2 in order to get the constrained types of e_1 and e_2 and then call the utilitary function subtype which will generate the additional subtyping constraints and dispatch the input and output types of the resulting expression.
  The structural condition that is checked immediately is that the number of outputs of e_1 be equal to the number of inputs of e_2.
  The additional subtyping constraint may be defined as follows, where (z_1,z_1') is the type of e_1 and (z_2,z_2') is the type of e_2:
  Let $c \in$ C, $\rho \in \mathcal{P}$(P) and $\iota \in \mathcal{P}$(I) so that $c = (\rho,\iota)$
  Then $\rho = \bigcup_{\substack{j\in[1,|z_2|] \\ (t_1',(l_1',u_1'))=z_1'(j) \\ (t_2,(l_2,u_2))=z_2(j)}} \{\, t_1' = t_2 \,\} \cup \{\, l_2 < l_1' \,\} \cup \{\, u_1' < u_2 \,\}$

  Here, t stands for a base type, l for a lower bound on an interval and u for an upper bound on an interval.
  The generation of this constraint is handled by the typing_predicate function, which may be used with either the $\subset$ operator for subtyping or the = operator for type equality.
  This case corresponds to the most common situation we will encounter in our input Faust expression, as other cases may be translated as an equivalent $e_1 : e_2$ case.

- In case $e = e_1 , e_2$, the beams of e_1 and the beams of e_2 run in parallel so this case is equivalent to appending the input beams of e_1 to the input beams of e_2 and the output beams of e_1 to the output beams of e_2.
  We first recursively call constrained_type on e_1 and e_2 in order to get the constrained types of e_1 and e_2 .
  There are no structural conditions to check, as the beams of e_1 and e_2 do not interact with one another.
  Then, we simply concatenate the constraints, the input beams and the output beams of the two expressions.
  This case is the easiest to handle, as it does not generate additional predicates to store in our global constraint.

- In case $e = e_1$ <: e_2 , each output signal of e_1 is transferred to multiple inputs of e_2. We first recursively call constrained_type on e_1 and e_2 in order to get the constrained types of e_1 and e_2.
  Structural constraints are checked immediately after that. They ensure that the two beams z_1' and z_2, which connect, are not null and that the length of z_2 is a multiple of the length of z_1', thus making sure that each signal from z_2 will receive an input from z_1'.
  By using the function split_to_seq, we then move from a split case to an equivalent $e_1 : e_2$ case, which will be directly handled by a new call to constrained_type. This way, we have indeed the $e_1 : e_2$ case serve as a translation target in the more complex cases of this algorithm.

- In case $e = e_1$:> e_2 , each input signal of e_2 receives a sum of outputs from e_1. We first recursively call constrained_type on e_1 and e_2 in order to get the constrained types of e_1 and e_2. Structural constraints are checked immediately after that.
  They ensure that the two beams z_1' and z_2, which connect, are not null and that the length of z_1' is a multiple of the length of z_2, thus making sure that each signal from z_1' will have a destination in z_2.
  By using the sum function, we then apply the function subtype as we did during the $e_1 : e_2$ case. In the end, we have indeed summed the output signals of e_1 into the corresponding inputs of e_2.

- In case $e = e_1 \sim e_2$, $|z_2|$ signals of beam z_1' are linked to beam z_2 and $|z_2'|$ signals of beam z_2' are linked to z_1. This way, a loop is created.
  We first recursively call constrained_type on e_1 and e_2 in order to get the constrained types of e_1 and e_2.
  Structural constraints are checked immediately after that. We check on the two sets of connexions, respectively at the input and output of e_1.
  The generation of the resulting constrained type is then handled by the function loop_equality, which handles what is basically the equivalent of two calls of the $e_1 : e_2$ case and a call to widening.

I use widening for the moment, but my goal is to use a better approximation through abstract interpretation, first basing myself on works like [23].

All the aforementioned elements yield the following outline for our algorithm.

```
constrained_type: R × C ⟶ E ⟶ K + fail

constrained_type ($r$,$c$) $e$ = **case** $e$ **in**

$i$ =>
    if $i \in$ dom $r$ then
        let $(t, l) = r\ i$
        let $l'$ = map new_unification_variable $l$
        let $t'$ = map ($\lambda xy$ (substitution $x\ y\ t$)) $l\ l'$
        ( $t'$, $(\varnothing, l') \cup c$ )
    else fail

$e_1$ : $e_2$ =>
    let ( (($z_1$,$z'_1$),$c'_1$) , (($z_2$,$z'_2$),$c'_2$) ) = map (constrained_type ($r$,$c$)) ( $e_1$ , $e_2$ )
    if $|z'_1| = |z_2|$ then
        subtype ($c'_1 \cup c'_2$) ($z'_1$,$z_2$)
    else fail

$e_1$ , $e_2$ =>
    let ( (($z_1$,$z'_1$),$c'_1$) , (($z_2$,$z'_2$),$c'_2$) ) = map (constrained_type ($r$,$c$)) ( $e_1$ , $e_2$ )
    ( (append $z_1\ z_2$, append $z'_1\ z'_2$) , $c'_1 \cup c'_2$ )

$e_1$ <: $e_2$ =>
    let ( (($z_1$,$z'_1$),$c'_1$) , (($z_2$,$z'_2$),$c'_2$) ) = map (constrained_type ($r$,$c$)) ( $e_1$ , $e_2$ )
    if ( $|z'_1||z_2| \neq 0$ & $|z_2|\%|z'_1| = 0$ ) then
        constrained_type ( $r$ , $c'_1 \cup c'_2$ ) ( split_to_seq ( $e_1$ , $e_2$ ) )
    else fail

$e_1$ :> $e_2$ =>
    let ( (($z_1$,$z'_1$),$c'_1$) , (($z_2$,$z'_2$),$c'_2$) ) = map (constrained_type ($r$,$c$)) ( $e_1$ , $e_2$ )
    if ($|z'_1||z_2| \neq 0$ & $|z'_1|\%|z_2| = 0$ ) then
        constrained_type ( $r$ , $c'_1 \cup c'_2$ ) ( merge_to_seq ( $e_1$ , $e_2$ ) )
    else fail

$e_1 \sim e_2$ =>
    let ( (($z_1$,$z'_1$),$c'_1$) , (($z_2$,$z'_2$),$c'_2$) ) = map (constrained_type ($r$,$c$)) ( $e_1$ , $e_2$ )
    if $|z_1| \geq |z'_2|$ & $|z'_1| \geq |z_2|$ then
        loop_approximation ($c'_1 \cup c'_2$) (($z_1$,$z'_1$),($z_2$,$z'_2$))
    else fail

else fail
```

3.2.3 Lower level functions

We shall now take a closer look at the functions that correspond to a lower level of abstraction that we used in the constrained type computation. Their contents are not expanded in the main algorithm because it belongs to a higher level of abstraction, which deals with higher order operations. These functions are presented in their order of appearance in the constrained_type function.

A particular point of interest is the loop_approximation function, in which I plan later on to insert an approximation based on abstract interpretation, instead of the widening approximation currently in place. Other points of interest are the split_to_seq and merge_to_seq functions, which respectively convert split and merge expressions into equivalent sequential expressions.

We start our list with the functions corresponding to the $e = i$ case of constrained_type.

new_unification_variable : This function is called in the $e = i$ case of the constrained_type function, where it is used as an argument of map. It simply creates a new unificaton variable, which is actually an identifier, by calling the constructor of the Identifier class. In our case, map is applied to the list l of variables of a type scheme so that it generates a new list l' where each element is the output of a call to new_unification_variable. Thus, l' is the list of unification variables instantiating the variables of the type scheme mentioned above.

substitution : This function is called in the $e = i$ case of the constrained_type function, where it is used as part of an argument of map. It replaces all occurences of identifier i_1 by identifier i_2 in type t. This way, it allows to instantiate the type scheme variable i_1 by unification variable i_2.

We now continue with the functions related to the $e_1 : e_2$ case of our algorithm.

subtype : This function is called in the $e_1 : e_2$ case of the constrained_type function. It applies the function typing_predicate before merging the generated constraint with the current constraint given as input. Here, constrained_type is used in the case of subtyping, hence the presence of the < operator as input.

Now, we can directly move on to the e_1 <: e_2 case of the constrained_type function.

split_to_seq : This function is called in the e_1 <: e_2 case of the constrained_type function. It converts e_1 <: e_2 into an equivalent $e_1' : e_2'$ expression. This conversion is mainly done by duplicating the signals of e_1 inside a bigger expression. The constraints are also reorganized accordingly.

Similarly, we now deal with the e_1 :> e_2 case of our algorithm.

merge_to_seq : This function is called in the e_1 :> e_2 case of the constrained_type function. It converts e_1 :> e_2 into an equivalent $e_1' : e_2'$ expression. This conversion is mainly done by summing the elements of e_1 into a smaller expression.The constraints are also reorganized accordingly. It requires to modify the boundaries of the related intervals and to pay attention to subtyping, which may occur at any time.

Finally, we deal with the functions that are called in the $e_1 \sim e_2$ part of the constrained_type function.

loop_approximation : This function is called in the $e_1 \sim e_2$ case of the constrained_type function. It enables us to use typing_predicate multiple times before actually merging the generated constraints with the current constraint given as input. Here, typing_predicate is used in the case of type equality.

widening : This function is called in the loop_approximation function. It widens the interval of each signal of a beam to]-∞,+∞[. This is the function I wish to optimize through the use of abstract interpretation.

3.3 Correctness theorems

The type_checking algorithm comes with both soundness and completeness conjectures. One of my current areas of study is to prove these conjectured theorems.

3.3.1 Soundness

If the type_checking algorithm returns a type t for the expression e, then the typing rules allow the definition of this type t for expression e.

Theorem 1. ***Soundess of the type_checking algorithm***

Let $e \in E,\ r \in R$ *and* $c \in C$

If constrained_type (r,c) e = (t,c′)

and solve c′ = m

Then $r \vdash e : m\ t$

3.3.2 Completeness

If the typing rules allow the definition of type t for expression e, then the type_checking algorithm either returns this type t or a subtype of t for expression e.

Theorem 2. ***Completeness of the type_checking algorithm***

Let $e \in E$, $r \in R$, $c \in C$ *and* $t \in T$

If $r \vdash e : t$ *and* *solve* $c \neq$ *fail*

Then $\exists\, m \in M$ *such that*

constrained_type $(r,c)\ e = (t',c')$, *solve* $c' = m$ *and* $t \supset m\, t'$

4 Current work and study perspectives

4.1 Completion of the current algorithm

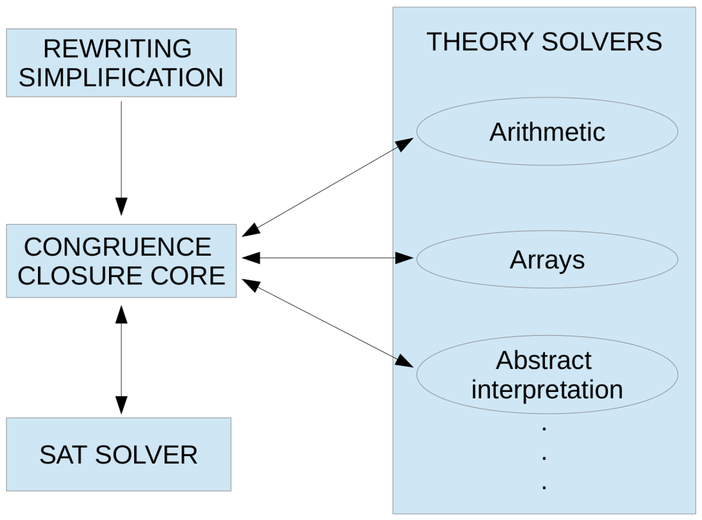
4.1.1 Solver

The solver part of the algorithm is currently being designed, based mainly on the Z3 [15] and Gecode [18] solvers. Other works which may prove to be of interest in this study are the veriT [3] and Redlog [6] solvers. The structure of these projects shall serve as the general model for the solve function of the type_checking algorithm. Practically, the function solve takes as input a constraint and either returns a model or fails, thus yielding the following signature:

solve : C $\longrightarrow$ T + fail

Here, the use of fail shall be more developped, as there is a difference to be pointed out between unsatisfiable constraints systems, systems whose satisfiability cannot be determined and the other usual occurences of failure .

All in all, the idea here is to create a lighter solver using only the theories that are required in our case of study. Thus, I would still use the congruence closure core of these solver models so that the related SAT solver may call the theories it needs through this core module. This would yield the following architecture.



This architecture mainly follows the models mentioned above, but with an abstract interpretation theory to be added to already existing theories.

A first step for the related implementation work would be to give the resulting constraint of the constrained_type function as an input to existing solvers. The constraints of my algorithm are thus to be translated into smt-lib [2] so that they may be handled by the existing solvers such as Z3. This way, the solver part of the main algorithm would be handled externally at first, thus enabling me to get a working global implementation that would be refined and made totally autonomous in a second time.

4.1.2 Proofs of the theorems

The proofs of the correctness theorems are also being designed at this moment. They are based on induction on the input expression, following the proofs presented in [5] and [9]. I currently admit the correctness of the solver part of the algorithm in order to already provide proofs involving the prior constraints generation part. Once the solver part of the algorithm is designed, I shall try to prove the correctness of some specific cases of this part as well.

Moreover, there are additional tracks of study for refining and extending these theorems. The following points are possible improvements I could apply to the theorems presented above. The main idea is to enhance the reliability of the type inference algorithm by avoiding to cast away solutions that were in reality acceptable.

Another approach to the soundness theorem is the following.

$\forall t \in T \;/\; e : t, \exists\, s \;/\; t = s(t^*)$

where s is a substitution from type to type and t^* is the type assigned to e by the typing rules.

What this formula states is that there exist substitutions allowing an expression e to have types that are equivalent to the one assigned by the typing rules. This case of study would provide additional decidable cases by linking equivalent types through substitution. This way, types that would have been ruled out for not being equal to t^* may be considered. Subtypes provide such an example.

Another approach to the completeness theorem is to replace $t \supset \mathrm{m}\ t'$ by $t = \mathrm{m}\ t'$ through the use of constrained types. With constraints on the intervals, it would be possible to force t to have the same interval as $\mathrm{m}\ t_0$. The base types being equal due to the constraints and conditions encountered during the type inference algorithm, this new approach would thus yield type equality instead of just subtyping.

4.2 New elements for the current algorithm

4.2.1 Multirate Faust

The current version of this project deals with the actual monorate version of Faust, and its very next goal is to then handle its multirate version, which was introduced in [10], [11] and [12]. In the very near future, I shall expand our algorithm to the multirate version of Faust. This will lead me to consider additional constraints concerning rates, as well as vector structures. Faust types are extended with information on the signal rate.

The implementation of the multirate version of Faust is currently under way at GRAME, and a first implementation has been made available on the SourceForge account of Faust. A first prototype of this multirate implementation is also available inside the Faustine interpreter.

4.2.2 Completion of the implementation

An implementation in OCaml, using the ground work laid by the Faustine interpreter [1], is currently being developped. This is a first prototype that will then be translated into C++ in order to be inserted inside the actual Faust compiler, so that it may be made available along with the official Faust release.

Based on this type checking algorithm, and using the classes infrastructure that is introduced throughout this document, it takes a Faust expression as input and outputs its type if it runs successfully. Right now, the algorithm and its implementation are handling the monorate version of Faust. My goal is now to have it handle the multirate case as well.

4.3 Further improvements

4.3.1 Use of abstract interpretation

Including an abstract interpretation aspect is part of the current tracks of studies for this algorithm. This would enable me to optimize the handling of the loop case of the Faust syntax. In order to do so, I shall use an approximation based on abstract interpretation instead of the current approximation based on widening. A particular notion of interest is to define a fixed point operator for use in this particular case.

In addition to inserting this construction inside the typing rules, I am currently studying two additional different approaches, which are to :

- insert this construction outside of the typing rules and directly set it into the constrained type generation part of the algorithm,
- insert an abstract interpretation theory in the theories module of the solver.

In order to get a better grasp of the developments already made available, I am currently studying existing works on abstract semantics for type systems, especially in the case of static type systems.

4.3.2 Integration into the Faust compiler

Currently, my implementation work has been dedicated to a prototype in OCaml of the type checking algorithm. This implementation currently relies on external solvers and is planned to be extended to the multirate version of Faust. Once these points are dealt with, this prototype in OCaml is to be converted to C++ and integrated into the Faust compiler. In order to achieve this, I shall of course translate the OCaml code into an equivalent C++ implementation, but I shall also implement my own solver instead of relying on external tools.

5 Additional prospective approaches

5.1 Polychronicity

A main further area of study to optimize the compiler of the Faust programming language is synchronicity, and especially polychronicity. Polychronous systems may have multiple clocks, which will be useful in the case of the multirate version of Faust. Such systems are the focus of the Polychrony framework [13], developped at IRISA. The main idea in the case of my thesis is to have a separate clock for every single frequency appearing in a Faust program.

The study of polychronous systems is recurrent in many research projects related to synchronous programming. It is a global theme that will surely be dealt with, no matter which particular synchronicity research subject I focus on.

5.2 Additional synchronicity studies

There are also numerous additional branches of study in the synchronous world such as :

- end-to-end latency,
- sequential constructivity,
- synchronous concurrency.

End-to-end latency provides constraints on synchronous systems. Sequential constructivity would enable me to set up an internal clock in Faust in order to manage input streams with different frequencies. Sequentially constructive concurrency would give me the tools to make constructive semantics for Faust.

I am studying several current areas of research so that I may find the most interesting domains of work for my PhD thesis. Currently, I am gathering global information on these subjects so that I may make informed decisions concerning the tracks I want my thesis to focus on.

5.3 Other tracks of study

In a general manner, typing rules are implicitly used in the current code of Faust but with no specification. In addition to the specifications I am already working on, it would be useful to insert the rewrite rules of Faust inside the typing system of Faust instead of having them directly implemented inside the compiler. It would also be interesting to try to prove the causality and termination of Faust programs. These are currently surmised. Such proofs could be made using Coq, and getting inspired from the proof of Compcert.

6 Doctoral courses and other skills

6.1 Scientific courses

Scientific courses have enabled me to get more acquainted with the current status of research on type systems and synchronicity, which are the two main areas of study realted to my PhD work.

The first class I took was a course on type systems and type inference at the “Master parisien de recherche en informatique” (MPRI): “Functional programming and type systems.“ It was organized by Xavier Leroy and Giuseppe Castagna, along with Didier Rémy and Yann Régis-Gianas. It is registered on my DOMINO account as 48 hours of scientific and technical courses.

The second class I took was a course on recent issues related to synchronicity and clocks at the “Collège de France”: “Enlarged time: multiple clocks, discrete times and continuous time.” It was organized by Gérard Berry and involved external lecturers as well. It is registered on my DOMINO account as 12 hours of scientific and technical courses.

These two courses are registered on my DOMINO account as 60 hours of scientific and technical courses, thus completing the 60 hours requirement for this category of courses.

6.2 Professionalizing courses

All the following courses have been provided by MINES ParisTech.

The first class I took was a course on bibliographical research: “Scientific publishing: strategies, tools and research optimization.” It is registered on my DOMINO account as 13 hours of professionalizing courses.

The second class I took was a course training me to promote my final PhD experience: “Point de départ.” It is registered on my DOMINO account as 14 hours of professionalizing courses.

The third class I am registered to is a course on fundamental law notions: “Practical law elements for life in a company.” This shall probably prove very useful further on. It will be registered on my DOMINO account as 30 hours of professionalizing courses.

These three courses will register on my DOMINO account as 57 (27 + 30) hours of professionalizing courses, thus leaving 3 remaining hours to fulfill.

In addition to these classes, the welcoming day of PhD students, providing global information about PhD organization, is also listed as a formation on my account.

6.3 English language proficiency

I got the following scores at the Toefl IBT : 103/120 (2009), 107/120 (2011). Official score reports for the Toefl IBT are available up to two years after the test, which is why I took the test in 2011 when the two-year period since 2009 was over.

In addition to these, I have a graduate degree from Columbia University in the City of New York : a Master of Science. I achieved it from 2011 to 2013 in the Computer Science Department of Columbia University. All my work and studies there were conducted in English.

7 Previsional planning

These are the previsional steps for the rest of my PhD thesis, which show how I plan to organize the completion of the points I presented earlier in this report.

- **Step 1 : 2014**
  - Final type checking algorithm handling multirate Faust and vectors, with a complete OCaml implementation.
  - Presentation for workshops.

- **Step 2 : 2014 - 2015**
  - Refinment of the widening by abstract interpretation.
  - Insertion of new synchronous developments.
  - Article for conferences.

- **Step 3 : 2015 - 2016**
  - Integration in the C++ compiler of Faust.
  - Delivery of the PhD thesis.

8 Conclusion

Currently, the main scientific contribution of my PhD work is the type checking algorithm of the Faust programming language. Through the design of this algorithm, I am providing a static typing system for Faust, thus optimizing its compiler.

In its present form, this typing algorithm is actually handling the monorate version of Faust and will soon handle the multirate version as well. This extension will require an update of the specifications so that they may handle the presence of a frequency field in Faust types.

My immediate work is to finish the prototype of the type checking algorithm in OCaml. In order to do so, I am currently completing the implementation of the monorate version of type_checking. Once this is done, I shall include vector structures and constraints on rates.

Longer term perspectives are focusing on the inclusion of abstract interpretation in the algorithm and on additional synchronous developments such as polychronicity. My ultimate goal is to integrate the type checking algorithm into the C++ compiler of Faust itself, thus making it an integral part of the Faust programming language.

In parallel of all this work, I have made presentations of my PhD thesis status during the meetings of the FEEVER project and at the IRISA center of INRIA. There is also a presentation on liquid types that is available online. I had presented it during the monthly seminars of CRI.

Eventually, I plan to finish the OCaml prototype by the end of 2014, while continuing the study of developments based on abstract interpretation and synchronicity until the end of 2015. Once these points have been treated, I shall insert this type checker into the actual compiler of Faust and deliver my PhD thesis.

9 Bibliography

[1] Karim Barkati, Haisheng Wang, and Pierre Jouvelot. Faustine: a vector faust interpreter test bed for multimedia signal processing. In *Twelfth International Symposium on Functional and Logic Programming (FLOPS 2014)*, 2014.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[3] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: an open, trustable and efficient SMT-Solver. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, number 5663 in Lecture Notes in Computer Science, pages 151–156. Springer Berlin Heidelberg, January 2009.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, page 238–252. ACM, 1977.

[5] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 207–212. ACM, 1982.

[6] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *Acm Sigsam Bulletin*, 31(2):2–9, 1997.

[7] Etienne Gaudrain and Yann Orlarey. A faust tutorial. Technical report, GRAME, Lyon, 2003.

[8] Andrew D Gordon and Cédric Fournet. Principles and applications of refinement types. *Logics and Languages for Reliability and Security*, 25, 2010.

[9] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 303–310. ACM, 1991.

[10] Pierre Jouvelot and Yann Orlarey. Semantics for multirate faust. *New Computational Paradigms for Computer Music-Editions Delatour France*, 2009.

[11] Pierre Jouvelot and Yann Orlarey. Dependent vector types for multirate faust. In *Proceedings of the Sound and Music Computing Conference*. Citeseer, 2010.

[12] Pierre Jouvelot and Yann Orlarey. Dependent vector types for data structuring in multirate faust. *Computer Languages, Systems & Structures*, 37(3):113–131, 2011.

[13] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(03):261–303, 2003.

[14] Kenneth L. McMillan and Andrey Rybalchenko. Solving constrained horn clauses using interpolation. Technical report, Technical Report MSR-TR-2013-6, Microsoft Research, 2013.

[15] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 4963 in Lecture Notes in Computer Science, pages 337–340. Springer Berlin Heidelberg, January 2008.

[16] Yann Orlarey, Dominique Fober, and Stephane Letz. FAUST: an efficient functional approach to DSP programming. *New Computational Paradigms for Computer Music*, 2009.

[17] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, page 159–169. ACM, 2008.

[18] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. *Software download and online material at the website: http://www. gecode. org*, 2006.

[19] Jean-Pierre Talpin, Jens Brandt, Mike Gemünde, Klaus Schneider, and Sandeep Shukla. Constructive polychronous systems. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, number 7734 in Lecture Notes in Computer Science, pages 335–349. Springer Berlin Heidelberg, January 2013.

[20] Olivier Tardieu, Nathaniel Nystrom, Igor Peshansky, and Vijay Saraswat. Constrained kinds. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 811–830, New York, NY, USA, 2012. ACM.

[21] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In *Programming Languages and Systems*, page 209–228. Springer, 2013.

[22] Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Bjorn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. Sequentially constructive concurrency a conservative extension of the synchronous model of computation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 581–586, March 2013.

[23] Jerome Vouillon and Pierre Jouvelot. *Type and Effect Systems via Abstract Interpretation*. Ecole des mines de Paris, Centre de recherche en informatique, 1995.

[24] Rémy Wyss, Frédéric Boniol, Claire Pagetti, and Julien Forget. End-to-end latency computation in a multi-periodic design. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, page 1682–1687, New York, NY, USA, 2013. ACM.

[25] Hongwei Xi. Dependent ML an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(02):215–286, 2007.

[26] Bin Xue and Sandeep K. Shukla. Modeling and analyzing the implementation of latency-insensitive protocols using the polychrony framework. *Electronic Notes in Theoretical Computer Science*, 245:3–22, 2009.