Poing Impératif: Compiling Imperative and Object Oriented Code to Faust

Kjetil Matheussen

Norwegian Center for Technology in Music and the Arts (NOTAM)

May 6, 2011

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

About Faust (background info)

- Faust is a programming language
  - ...for making programs which process audio signals.
- High level language.
  - Code is more compact and cleaner than C or C++.
  - Less fiddling with details. (less bugs and easier to read)
- Faust generates very efficient code.
  - Often competes with handwritten C++ code.
  - Faust can optimize code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked in the C or C++ version.
- Automatically generates various formats such as LADSPA, VST, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, OpenCL, etc.
  - Write once, run everywhere. (even on your gfx board!)
  - Options for generating parallel code. (automatically)
  - Option for generating code which are more easily vectorized. (I.e. to generate SIMD assembler instructions.)
- Conclusion: Many advantages of using Faust instead of C or C++.

Description of the problem

1. Faust requires the programmer to immediately start thinking in fully functional terms.
   - A 400Hz sine oscillator can not be made like this in faust:

     ```
     phase = 0.0;
     process(){
       phase = phase + 400*(pi*2/samplerate);
       return sin(phase);
     }
     ```

   - A special recursive operator (tilde) must be used instead:

     ```
     process = _ ~ +(400*(pi*2/samplerate)) : sin;
     ```

2. Not straight forward to translate DSP code written in C or C++ to Faust. (because of fundamentally different programming paradigms)

Description of the problem

1. Faust requires the programmer to immediately start thinking in fully functional terms.
   - A 400Hz sine oscillator can not be made like this in faust:

     ```
     phase = 0.0;
     process(){
       phase = phase + 400*(pi*2/samplerate);
       return sin(phase);
     }
     ```

   - A special recursive operator (tilde) must be used instead:

     ```
     process = _ ~ +(400*(pi*2/samplerate)) : sin;
     ```

2. Not straight forward to translate DSP code written in C or C++ to Faust. (because of fundamentally different programming paradigms)

Description of the problem

1. Faust requires the programmer to immediately start thinking in fully functional terms.
   - A 400Hz sine oscillator can not be made like this in faust:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

   - A special recursive operator (tilde) must be used instead:

```
process = _ ~ +(400*(pi*2/samplerate)) : sin;
```

2. Not straight forward to translate DSP code written in C or C++ to Faust. (because of fundamentally different programming paradigms)

Description of the problem

1. Faust requires the programmer to immediately start thinking in fully functional terms.
   - A 400Hz sine oscillator can not be made like this in faust:

     ```
     phase = 0.0;
     process(){
       phase = phase + 400*(pi*2/samplerate);
       return sin(phase);
     }
     ```

   - A special recursive operator (tilde) must be used instead:

     ```
     process = _ ~ +(400*(pi*2/samplerate)) : sin;
     ```

2. Not straight forward to translate DSP code written in C or C++ to Faust. (because of fundamentally different programming paradigms)

Description of the problem

1. Faust requires the programmer to immediately start thinking in fully functional terms.
   - A 400Hz sine oscillator can not be made like this in faust:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

   - A special recursive operator (tilde) must be used instead:

```
process = _ ~ +(400*(pi*2/samplerate)) : sin;
```

2. Not straight forward to translate DSP code written in C or C++ to Faust. (because of fundamentally different programming paradigms)

Solution to the problem

- A new compiler called Poing Impératif. This compiler
  - Extends Faust with imperative and object oriented features.
  - Outputs pure Faust code.
- Now, a 400Hz sine oscillator can be implemented like this:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

- Poing Impératif makes it easier to:
  1. Start using Faust without having to immediately start thinking in fully functional terms.
  2. Translate imperative and object oriented code to Faust.

Solution to the problem

- A new compiler called Poing Impératif. This compiler
  - Extends Faust with imperative and object oriented features.
  - Outputs pure Faust code.
- Now, a 400Hz sine oscillator can be implemented like this:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

- Poing Impératif makes it easier to:
  1. Start using Faust without having to immediately start thinking in fully functional terms.
  2. Translate imperative and object oriented code to Faust.

Solution to the problem

- A new compiler called Poing Impératif. This compiler
  - Extends Faust with imperative and object oriented features.
  - Outputs pure Faust code.
- Now, a 400Hz sine oscillator can be implemented like this:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

- Poing Impératif makes it easier to:
  1. Start using Faust without having to immediately start thinking in fully functional terms.
  2. Translate imperative and object oriented code to Faust.

Solution to the problem

- A new compiler called Poing Impératif. This compiler
  - Extends Faust with imperative and object oriented features.
  - Outputs pure Faust code.
- Now, a 400Hz sine oscillator can be implemented like this:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

- Poing Impératif makes it easier to:
  1. Start using Faust without having to immediately start thinking in fully functional terms.
  2. Translate imperative and object oriented code to Faust.

Solution to the problem

- A new compiler called Poing Impératif. This compiler
  - Extends Faust with imperative and object oriented features.
  - Outputs pure Faust code.
- Now, a 400Hz sine oscillator can be implemented like this:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

- Poing Impératif makes it easier to:
  1. Start using Faust without having to immediately start thinking in fully functional terms.
  2. Translate imperative and object oriented code to Faust.

Solution to the problem

- A new compiler called Poing Impératif. This compiler
  - Extends Faust with imperative and object oriented features.
  - Outputs pure Faust code.
- Now, a 400Hz sine oscillator can be implemented like this:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

- Poing Impératif makes it easier to:
  1. Start using Faust without having to immediately start thinking in fully functional terms.
  2. Translate imperative and object oriented code to Faust.

Solution to the problem

- A new compiler called Poing Impératif. This compiler
  - Extends Faust with imperative and object oriented features.
  - Outputs pure Faust code.
- Now, a 400Hz sine oscillator can be implemented like this:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

- Poing Impératif makes it easier to:
  1. Start using Faust without having to immediately start thinking in fully functional terms.
  2. Translate imperative and object oriented code to Faust.

Solution to the problem

- A new compiler called Poing Impératif. This compiler
  - Extends Faust with imperative and object oriented features.
  - Outputs pure Faust code.
- Now, a 400Hz sine oscillator can be implemented like this:

```
phase = 0.0;
process(){
  phase = phase + 400*(pi*2/samplerate);
  return sin(phase);
}
```

- Poing Impératif makes it easier to:
  1. Start using Faust without having to immediately start thinking in fully functional terms.
  2. Translate imperative and object oriented code to Faust.

Example 1. Oscillator

```
class Oscillator(float frequency){
    float phase;

    float process(){
        phase += frequency*3.14*2/44100;
        return sin(phase);
    }
}

freq = hslider("freq",400.0,10,3000,1);

process = Oscillator(freq);
```

Example 2. Oscillator with local method

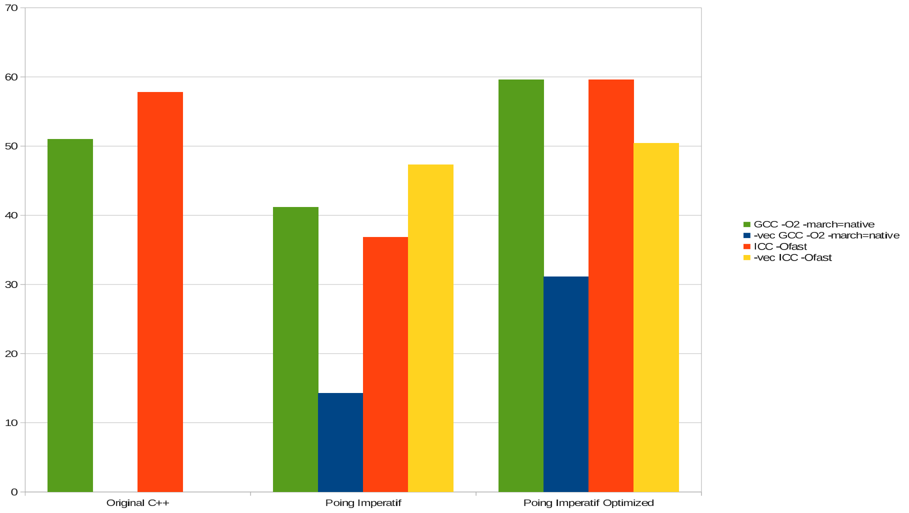
```
class Oscillator(float frequency){
  float phase;

  increase_phase(float how_much){
    phase += how_much;
  }

  float process(){
    this.increase_phase(frequency*3.14*2/44100);
    return sin(phase);
  }
}

freq = hslider("freq",400.0,10,3000,1);
process = Oscillator(freq);
```

Example 3. Oscillator using a separate Phase class

```
class Phase{
  float phase;

  increase_phase(float how_much){
    phase += how_much;
  }
}


class Oscillator(float frequency){
  Phase phase;

  float process(){
    phase.increase_phase(frequency*3.14*2/44100);
    return sin(phase.phase);
  }
}

freq = hslider("freq",400.0,10,3000,1);
process = Oscillator(freq);
```

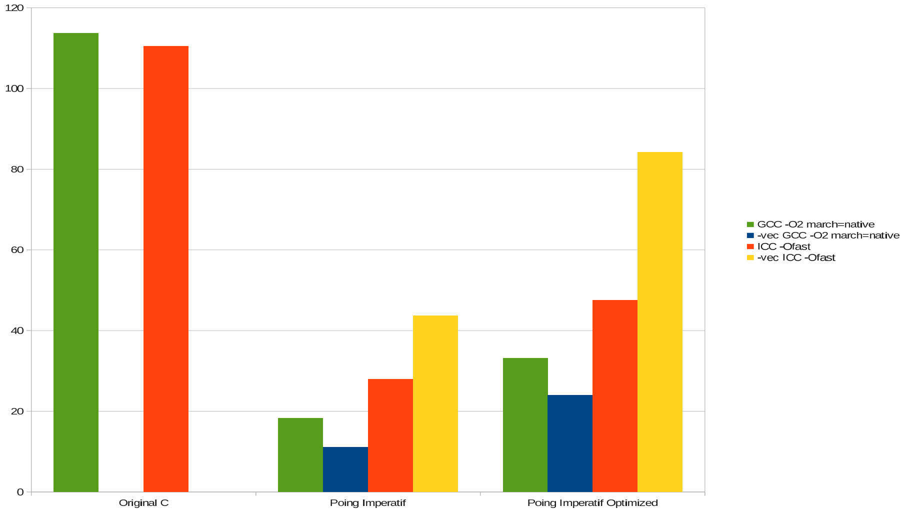
Example 4. Freeverb

Benchmark 1: Freeverb



Example 5. LADSPA am_pitchshift

Benchmark 2: LADSPA am_pitchshift



Limitations in Poing Impératif

1. Limited Array functionality
2. Missing *for* loop functionality
3. Inefficient branching

Limitations in Poing Impératif

1. Limited Array functionality
2. Missing *for* loop functionality
3. Inefficient branching

Limitations in Poing Impératif

1. Limited Array functionality
2. Missing *for* loop functionality
3. Inefficient branching

Limitations in Poing Impératif

1. Limited Array functionality
2. Missing *for* loop functionality
3. Inefficient branching

1. Limited Array functionality

- In C or C++ you can do this:

```
process(a,b){
  a[i]   += a;     // Statement 1
  a[i+1] += b;     // Statement 2
  return [i+2];
}
```

- But in Poing Impératif you can only do this:

```
process(a,b){
  a[i] += a;       // Statement 1
  return [i+2];
}
```

- or this:

```
process(a,b){
  a[i+2] += b;     // Statement 2
  return [i+2];
}
```

1. Limited Array functionality

- In C or C++ you can do this:

```
process(a,b){
  a[i]   += a;    // Statement 1
  a[i+1] += b;    // Statement 2
  return [i+2];
}
```

- But in Poing Impératif you can only do this:

```
process(a,b){
  a[i] += a;      // Statement 1
  return [i+2];
}
```

- or this:

```
process(a,b){
  a[i+2] += b;    // Statement 2
  return [i+2];
}
```

1. Limited Array functionality

- In C or C++ you can do this:

```
process(a,b){
  a[i]   += a;    // Statement 1
  a[i+1] += b;    // Statement 2
  return [i+2];
}
```

- But in Poing Impératif you can only do this:

```
process(a,b){
  a[i] += a;      // Statement 1
  return [i+2];
}
```

- or this:

```
process(a,b){
  a[i+2] += b;    // Statement 2
  return [i+2];
}
```

1. Limited Array functionality

- In C or C++ you can do this:

```
process(a,b){
  a[i]   += a;    // Statement 1
  a[i+1] += b;    // Statement 2
  return [i+2];
}
```

- But in Poing Impératif you can only do this:

```
process(a,b){
  a[i] += a;      // Statement 1
  return [i+2];
}
```

- or this:

```
process(a,b){
  a[i+2] += b;    // Statement 2
  return [i+2];
}
```

2. Missing *for* loop functionality

1. In C or C++ you can do this:

   ```
   int get_faculty(int len){
     int faculty = 1;
     for(int i=2; i<len; i++){
       faculty *= i;
     }
     return faculty;
   }
   ```

   This is not supported in Poing Impératif. (and is quite unlikely to be in the future.)

2. In C++ you can do this:

   ```
   #define LEN 50

   int get_faculty(){
     int faculty = 1;
     for(int i=2 ;i<LEN ;i++){
       faculty *= i;
     }
     return faculty;
   }
   ```

   This is not supported in Poing Impératif either. (but might be supported in the future.)

2. Missing *for* loop functionality

1. In C or C++ you can do this:

   ```
   int get_faculty(int len){
     int faculty = 1;
     for(int i=2; i<len; i++){
       faculty *= i;
     }
     return faculty;
   }
   ```

   This is not supported in Poing Impératif. (and is quite unlikely to be in the future.)

2. In C++ you can do this:

   ```
   #define LEN 50

   int get_faculty(){
     int faculty = 1;
     for(int i=2 ;i<LEN ;i++){
       faculty *= i;
     }
     return faculty;
   }
   ```

   This is not supported in Poing Impératif either. (but might be supported in the future.)

2. Missing *for* loop functionality

1. In C or C++ you can do this:

```
int get_faculty(int len){
  int faculty = 1;
  for(int i=2; i<len; i++){
    faculty *= i;
  }
  return faculty;
}
```

   This is not supported in Poing Impératif. (and is quite unlikely to be in the future.)

2. In C++ you can do this:

```
#define LEN 50

int get_faculty(){
  int faculty = 1;
  for(int i=2 ;i<LEN ;i++){
    faculty *= i;
  }
  return faculty;
}
```

   This is not supported in Poing Impératif either. (but might be supported in the future.)

3. Inefficient branching

1. Faust generate no jumps.
   Faust uses ?: as value selectors.
   (For instance $a = b\ ?\ 3 : 4;$)

2. Example:

```
if(a==1){
  lots of things 1.
}else{
  lots of things 2.
}
```

3. However, a very intelligent C compiler could create jumps out of ?: selectors.

3. Inefficient branching

1. Faust generate no jumps.
   Faust uses ?: as value selectors.
   (For instance $a = b\ ?\ 3 : 4;$)

2. Example:

```
if(a==1){
  lots of things 1.
}else{
  lots of things 2.
}
```

3. However, a very intelligent C compiler could create jumps out of ?: selectors.

3. Inefficient branching

1. Faust generate no jumps.
   Faust uses ?: as value selectors.
   (For instance $a = b\ ?\ 3 : 4;$)

2. Example:

```
if(a==1){
  lots of things 1.
}else{
  lots of things 2.
}
```

3. However, a very intelligent C compiler could create jumps out of ?: selectors.

3. Inefficient branching

1. Faust generate no jumps.
   Faust uses ?: as value selectors.
   (For instance $a = b\ ?\ 3 : 4;$)

2. Example:

```
if(a==1){
  lots of things 1.
}else{
  lots of things 2.
}
```

3. However, a very intelligent C compiler could create jumps out of ?: selectors.

Future work

- Implement *for* loops.
- Reduce compilation time.
  - Freeverb takes 20-40 seconds to compile.
  - Worse: small changes in the freeverb code causes Faust never to finish. (apparently)

Future work

- Implement *for* loops.
- Reduce compilation time.
  - Freeverb takes 20-40 seconds to compile.
  - Worse: small changes in the freeverb code causes Faust never to finish. (apparently)

Future work

- Implement *for* loops.
- Reduce compilation time.
  - Freeverb takes 20-40 seconds to compile.
  - Worse: small changes in the freeverb code causes Faust never to finish. (apparently)

Future work

- Implement *for* loops.
- Reduce compilation time.
  - Freeverb takes 20-40 seconds to compile.
  - Worse: small changes in the freeverb code causes Faust never to finish. (apparently)

Future work

- Implement *for* loops.
- Reduce compilation time.
  - Freeverb takes 20-40 seconds to compile.
  - Worse: small changes in the freeverb code causes Faust never to finish. (apparently)

Q/A

Any questions?

Bakgrunn og problem

Utgangspunkt: Ønsket å bruke Stalin Scheme eller Bigloo Scheme for å programmere lyd i sanntid.

- Stalin Scheme og Bigloo Scheme bruker Hans Boehm's konservative søppeltømmer for C og C++ (BDW-GC).

Problem 1: BDW-GC virker dårlig i sanntid.

- Programmene må vente til "mark" er ferdig å kjøre.

  Uberegnelig pausetid

Problem 2: BDW-GC krever ikke read barrier eller write barrier.

- Alle kjente sanntids-søppeltømmere krever read barrier eller write barrier.
- Stalin Scheme / Bigloo Scheme må i tilfelle modifiseres.

Bakgrunn og problem

Utgangspunkt: Ønsket å bruke Stalin Scheme eller Bigloo Scheme for å programmere lyd i sanntid.

- Stalin Scheme og Bigloo Scheme bruker Hans Boehm's konservative søppeltømmer for C og C++ (BDW-GC).

Problem 1: BDW-GC virker dårlig i sanntid.

- Programmene må vente til "mark" er ferdig å kjøre.

  Uberegnelig pausetid

Problem 2: BDW-GC krever ikke read barrier eller write barrier.

- Alle kjente sanntids-søppeltømmere krever read barrier eller write barrier.
- Stalin Scheme / Bigloo Scheme må i tilfelle modifiseres.

Bakgrunn og problem

Utgangspunkt: Ønsket å bruke Stalin Scheme eller Bigloo Scheme for å programmere lyd i sanntid.

- Stalin Scheme og Bigloo Scheme bruker Hans Boehm's konservative søppeltømmer for C og C++ (BDW-GC).

Problem 1: BDW-GC virker dårlig i sanntid.

- Programmene må vente til "mark" er ferdig å kjøre.
  - **Uberegnelig pausetid.**

Problem 2: BDW-GC krever ikke read barrier eller write barrier.

- Alle kjente sanntids-søppeltømmere krever read barrier eller write barrier.
- Stalin Scheme / Bigloo Scheme må i tilfelle modifiseres.

Bakgrunn og problem

Utgangspunkt: Ønsket å bruke Stalin Scheme eller Bigloo Scheme for å programmere lyd i sanntid.

- Stalin Scheme og Bigloo Scheme bruker Hans Boehm's konservative søppeltømmer for C og C++ (BDW-GC).

Problem 1: BDW-GC virker dårlig i sanntid.

- Programmene må vente til "mark" er ferdig å kjøre.
  - **Uberegnelig pausetid.**

Problem 2: BDW-GC krever ikke read barrier eller write barrier.

- Alle kjente sanntids-søppeltømmere krever read barrier eller write barrier.
- Stalin Scheme / Bigloo Scheme må i tilfelle modifiseres.

Bakgrunn og problem

Utgangspunkt: Ønsket å bruke Stalin Scheme eller Bigloo Scheme for å programmere lyd i sanntid.

- Stalin Scheme og Bigloo Scheme bruker Hans Boehm's konservative søppeltømmer for C og C++ (BDW-GC).

Problem 1: BDW-GC virker dårlig i sanntid.

- Programmene må vente til “mark” er ferdig å kjøre.
  - **Uberegnelig pausetid**.

Problem 2: BDW-GC krever ikke read barrier eller write barrier.

- Alle kjente sanntids-søppeltømmere krever read barrier eller write barrier.
- Stalin Scheme / Bigloo Scheme må i tilfelle modifiseres.

Bakgrunn og problem

Utgangspunkt: Ønsket å bruke Stalin Scheme eller Bigloo Scheme for å programmere lyd i sanntid.

- Stalin Scheme og Bigloo Scheme bruker Hans Boehm's konservative søppeltømmer for C og C++ (BDW-GC).

Problem 1: BDW-GC virker dårlig i sanntid.

- Programmene må vente til "mark" er ferdig å kjøre.
  - **Uberegnelig pausetid**.

Problem 2: BDW-GC krever ikke read barrier eller write barrier.

- Alle kjente sanntids-søppeltømmere krever read barrier eller write barrier.
- Stalin Scheme / Bigloo Scheme må i tilfelle modifiseres.

Bakgrunn og problem

Utgangspunkt: Ønsket å bruke Stalin Scheme eller Bigloo Scheme for å programmere lyd i sanntid.

- Stalin Scheme og Bigloo Scheme bruker Hans Boehm's konservative søppeltømmer for C og C++ (BDW-GC).

Problem 1: BDW-GC virker dårlig i sanntid.

- Programmene må vente til "mark" er ferdig å kjøre.
  - **Uberegnelig pausetid**.

Problem 2: BDW-GC krever ikke read barrier eller write barrier.

- Alle kjente sanntids-søppeltømmere krever read barrier eller write barrier.
- Stalin Scheme / Bigloo Scheme må i tilfelle modifiseres.

Bakgrunn og problem

Utgangspunkt: Ønsket å bruke Stalin Scheme eller Bigloo Scheme for å programmere lyd i sanntid.

- Stalin Scheme og Bigloo Scheme bruker Hans Boehm's konservative søppeltømmer for C og C++ (BDW-GC).

Problem 1: BDW-GC virker dårlig i sanntid.

- Programmene må vente til “mark” er ferdig å kjøre.
  - **Uberegnelig pausetid**.

Problem 2: BDW-GC krever ikke read barrier eller write barrier.

- Alle kjente sanntids-søppeltømmere krever read barrier eller write barrier.
- Stalin Scheme / Bigloo Scheme må i tilfelle modifiseres.