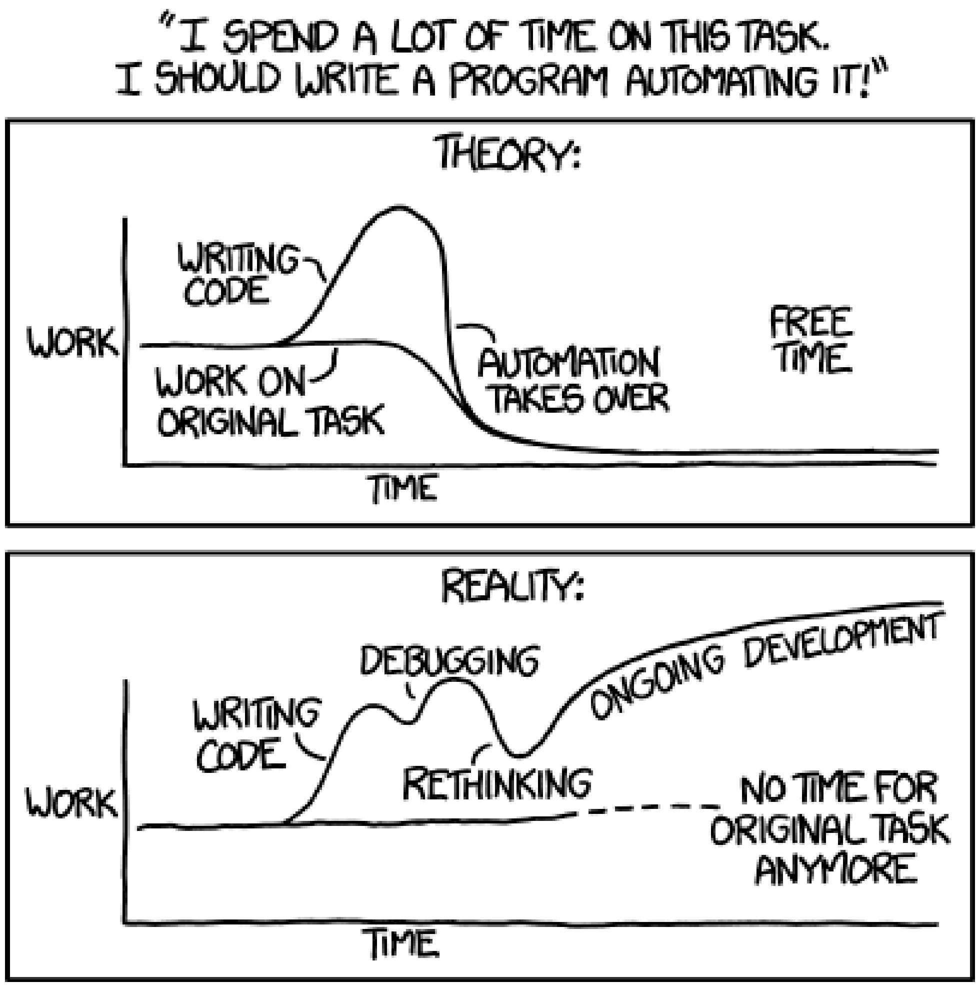
Language environments for building program generators

Georg Ofenbeck

Department of Computer Science
ETH Zürich, Switzerland

ETH zürich

SPIRAL
www.spiral.net



http://xkcd.com/1319/

Spiral

DFT(n)

Spiral

DFT(n)

$\mathbf{DFT}_4$

Spiral

DFT(n) $\xrightarrow{\text{decompose}}$ SPL

$\mathbf{DFT}_4$

Spiral

DFT(n) $\xrightarrow{\text{decompose}}$ SPL

$$\mathbf{DFT}_4$$

$$(\mathbf{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathbf{DFT}_2) L_2^4$$

Spiral

DFT(n) $\xrightarrow{\text{decompose}}$ SPL $\xrightarrow{\text{translate}}$ Σ-SPL

$\mathbf{DFT}_4$

$$(\mathbf{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathbf{DFT}_2) L_2^4$$

Spiral

$$\text{DFT}(n) \xrightarrow{\text{decompose}} \text{SPL} \xrightarrow{\text{translate}} \Sigma\text{-SPL}$$

$$\mathbf{DFT}_4$$

$$(\mathbf{DFT}_2 \otimes I_2) T^4_2 (I_2 \otimes \mathbf{DFT}_2) L^4_2$$

$$\left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j)\right) T^n_m \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j)\right) \mathrm{perm}(\ell^4_2)$$

Spiral

$$\text{DFT}(n) \xrightarrow{\text{decompose}} \text{SPL} \xrightarrow{\text{translate}} \Sigma\text{-SPL} \xrightarrow{\text{translate}} \text{C-IR}$$

$$\mathbf{DFT}_4$$

$$(\mathbf{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathbf{DFT}_2) L_2^4$$

$$\left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right) T_m^n \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right) \mathrm{perm}(\ell_2^4)$$

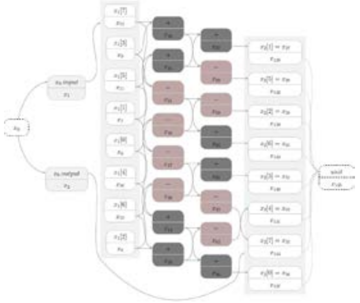
Spiral

$$\text{DFT(n)} \xrightarrow{\text{decompose}} \text{SPL} \xrightarrow{\text{translate}} \Sigma\text{-SPL} \xrightarrow{\text{translate}} \text{C-IR}$$

$$\mathbf{DFT}_4$$

$$(\mathbf{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathbf{DFT}_2) L_2^4$$

$$\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right) T_m^n \left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right) \mathrm{perm}(\ell_2^4)$$



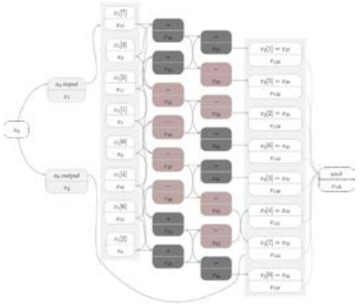
Spiral

$$\text{DFT(n)} \xrightarrow{\text{decompose}} \text{SPL} \xrightarrow{\text{translate}} \Sigma\text{-SPL} \xrightarrow{\text{translate}} \text{C-IR} \xrightarrow{\text{translate}} \text{C-Code}$$

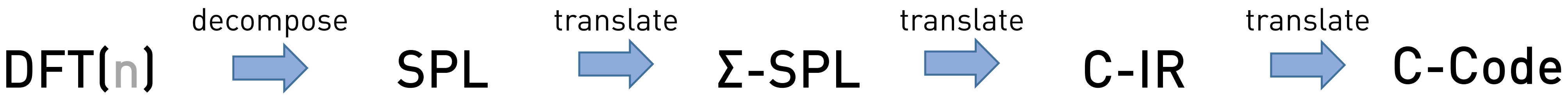
$$\mathbf{DFT}_4$$

$$(\mathbf{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathbf{DFT}_2) L_2^4$$

$$\left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right) T_m^n \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right) \mathrm{perm}(\ell_2^4)$$



Spiral

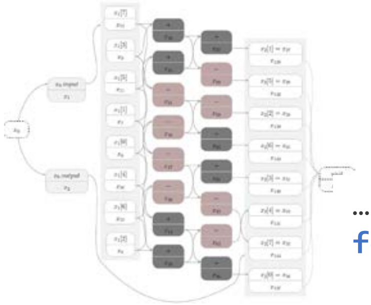


DFT(n) SPL Σ-SPL C-IR C-Code

$$\mathbf{DFT}_4$$

$$(\mathbf{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathbf{DFT}_2) L_2^4$$

$$\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right) T_m^n \left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right) \mathrm{perm}(\ell_2^4)$$



```
…
for (int i=0;i < 2;i++) {
   y[i]     = x[i] + x[i*2+1];
   y[i*2+1] = x[i] - x[i*2+1];
}
for (int i=0;i < 4;i++) {
   y[i] = y[i] * sin(PI/4*(i+2));
}
…
```

Spiral

$$\mathrm{DFT}(n) \xrightarrow{\text{decompose}} \mathrm{SPL} \xrightarrow{\text{translate}} \Sigma\text{-}\mathrm{SPL} \xrightarrow{\text{translate}} \text{C-IR} \xrightarrow{\text{translate}} \text{C-Code} \xrightarrow{\text{measure}}$$ ⏱

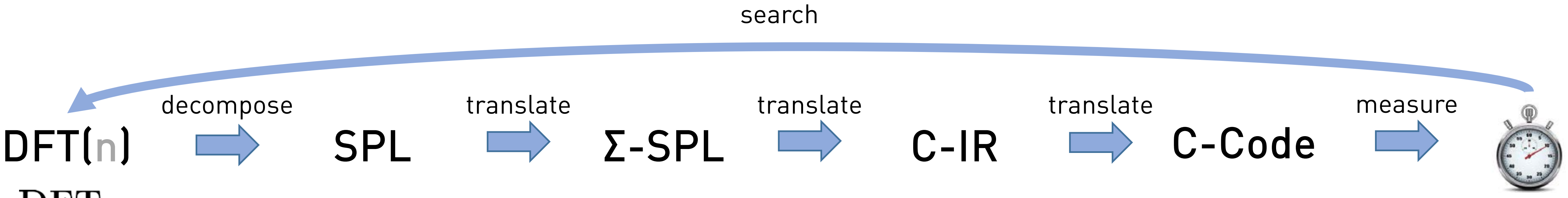
$$\mathbf{DFT}_4$$

$$(\mathbf{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathbf{DFT}_2) L_2^4$$

$$\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right) T_m^n \left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right) \mathrm{perm}(\ell_2^4)$$

```
…
for (int i=0;i < 2;i++) {
   y[i]     = x[i] + x[i*2+1];
   y[i*2+1] = x[i] - x[i*2+1];
}
for (int i=0;i < 4;i++) {
   y[i] = y[i] * sin(PI/4*(i+2));
}
…
```

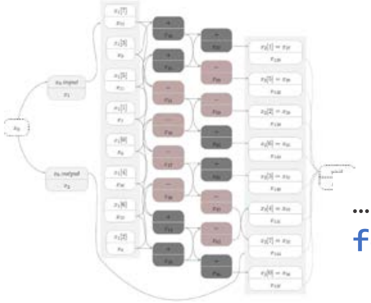
Spiral



$$\mathbf{DFT}_4$$

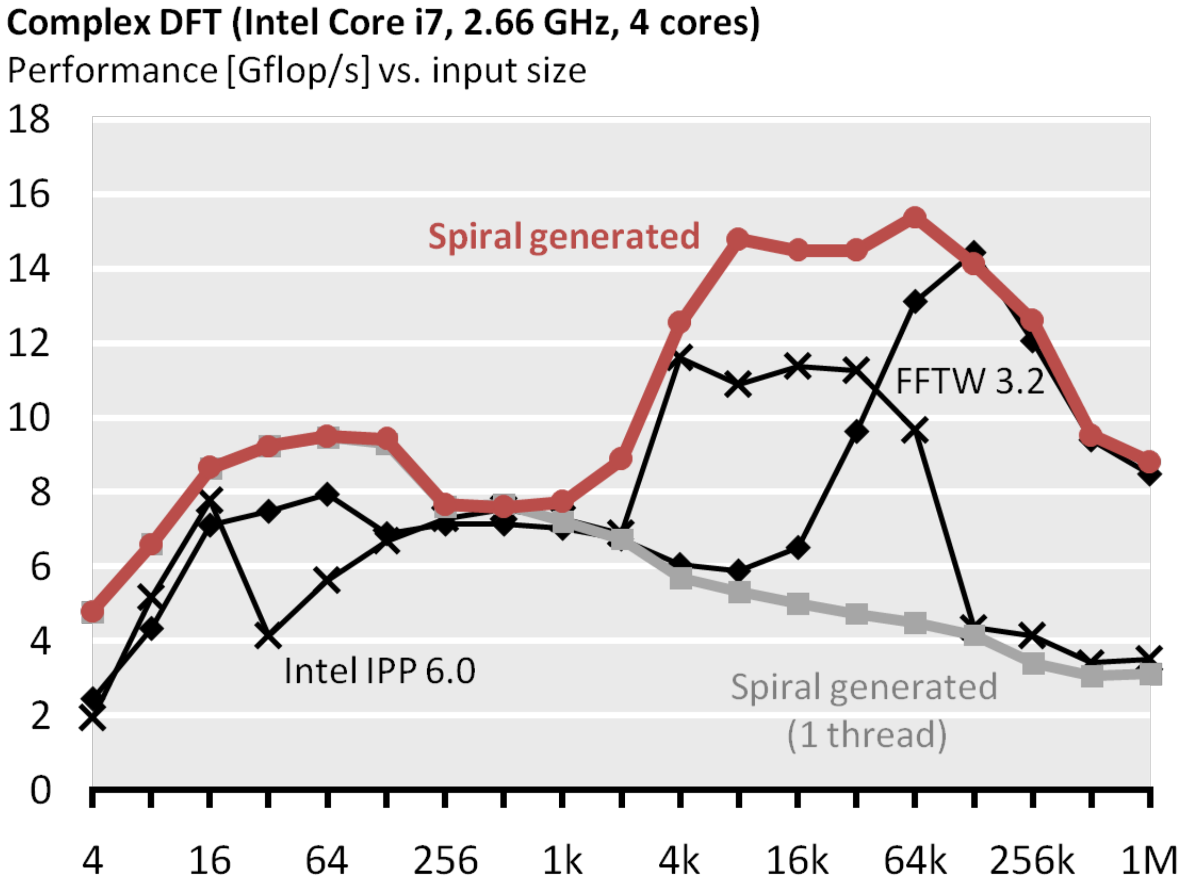
$$(\mathbf{DFT}_2 \otimes I_2) T^4_2 (I_2 \otimes \mathbf{DFT}_2) L^4_2$$

$$\left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right) T^n_m \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right) \mathrm{perm}(\ell^4_2)$$

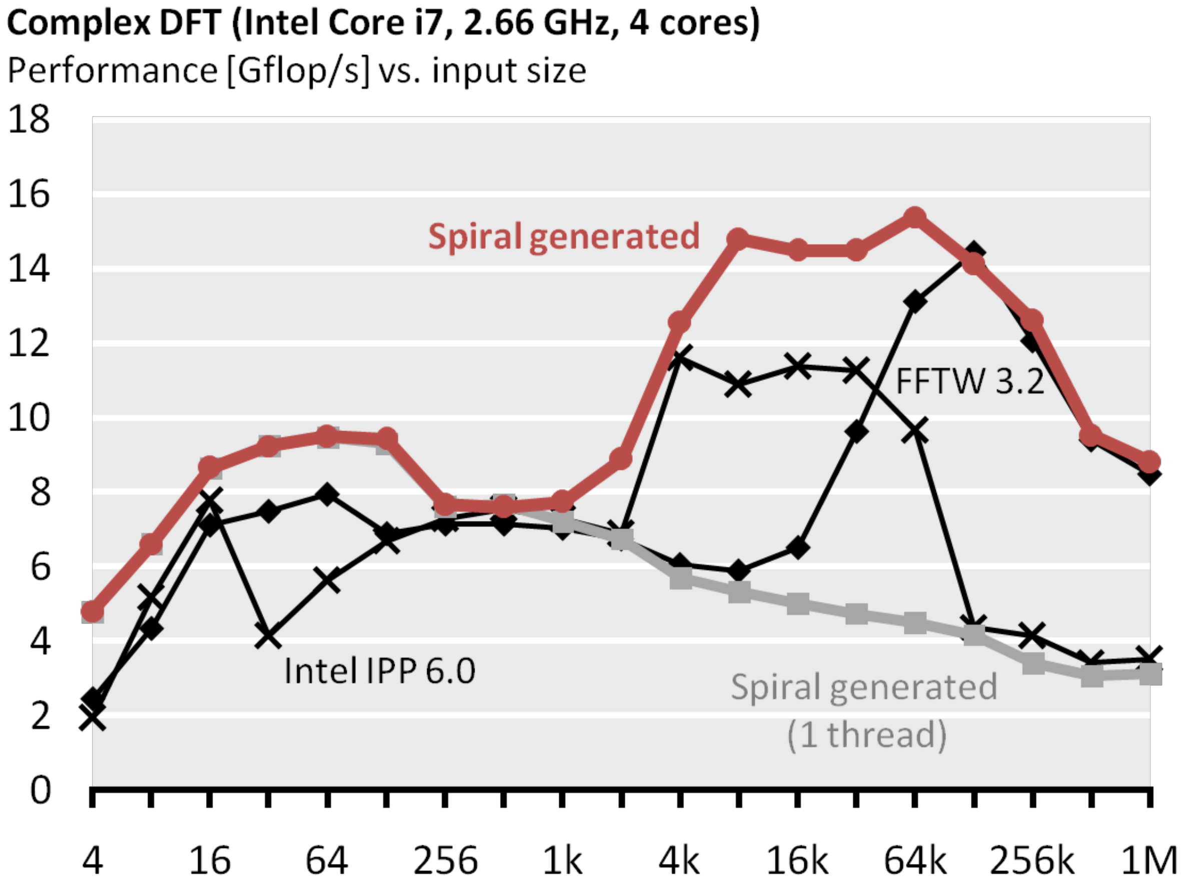


```
…
for (int i=0;i < 2;i++) {
   y[i]     = x[i] + x[i*2+1];
   y[i*2+1] = x[i] - x[i*2+1];
}
for (int i=0;i < 4;i++) {
   y[i] = y[i] * sin(PI/4*(i+2));
}
…
```

DFT on Intel Multicore



DFT on Intel Multicore



Complex DFT (Intel Core i7, 2.66 GHz, 4 cores)

Performance [Gflop/s] vs. input size

$$\mathbf{DFT}_n \rightarrow (\mathbf{DFT}_k \otimes I_m) T_m^n (I_k \otimes \mathbf{DFT}_m) L_k^n$$

$$\mathbf{DFT}_n \rightarrow P_{k/2,2m}^{\top} \left(\mathbf{DFT}_{2m} \oplus \left(I_{k/2-1} \otimes_i C_{2m}\, \mathbf{rDFT}_{2m}(i/k)\right)\right) (\mathbf{RDFT}_k \otimes I_m)$$

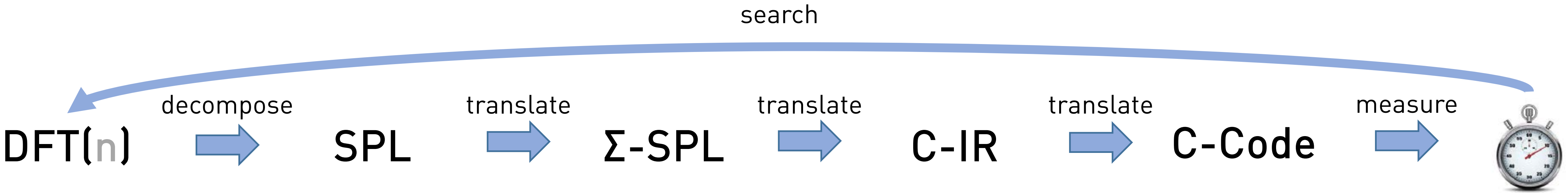
$$\mathbf{RDFT}_n \rightarrow (P_{k/2,m}^{\top} \otimes I_2) \left(\mathbf{RDFT}_{2m} \oplus \left(I_{k/2-1} \otimes_i D_{2m}\, \mathbf{rDFT}_{2m}(i/k)\right)\right) (\mathbf{RDFT}_k \otimes I_m)$$

$$\mathbf{rDFT}_{2n}(u) \rightarrow L_m^{2n} (I_k \otimes_i \mathbf{rDFT}_{2m}((i+u)/k)) (\mathbf{rDFT}_{2k}(u) \otimes I_m)$$

Spiral

5MB vectorized, threaded, general-size, adaptive library

My Research



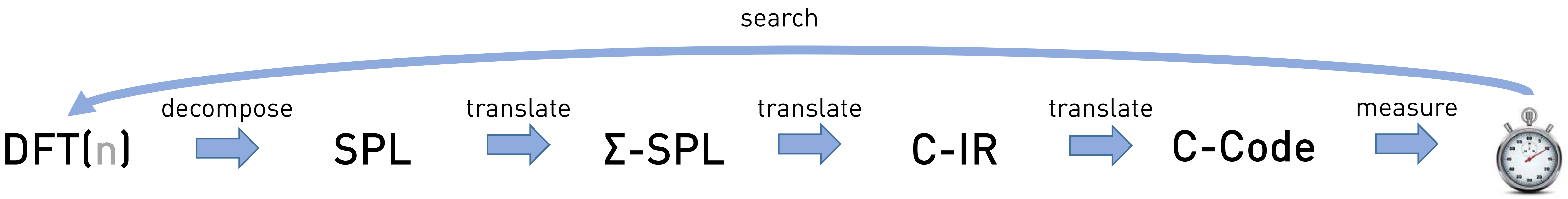
search

DFT(n) decompose SPL translate Σ-SPL translate C-IR translate C-Code measure

Can we utilize modern program language features for:

;

My Research



search

DFT(n) decompose SPL translate Σ-SPL translate C-IR translate C-Code measure

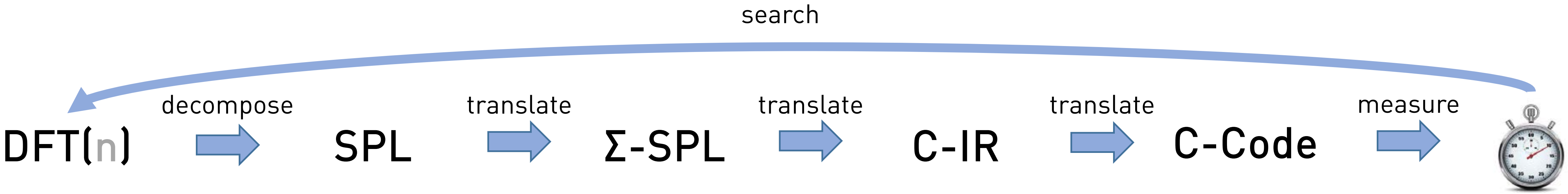
Can we utilize modern program language features for:

DSLs

- Translation between DSLs
- Rewrites on DSLs

;

My Research



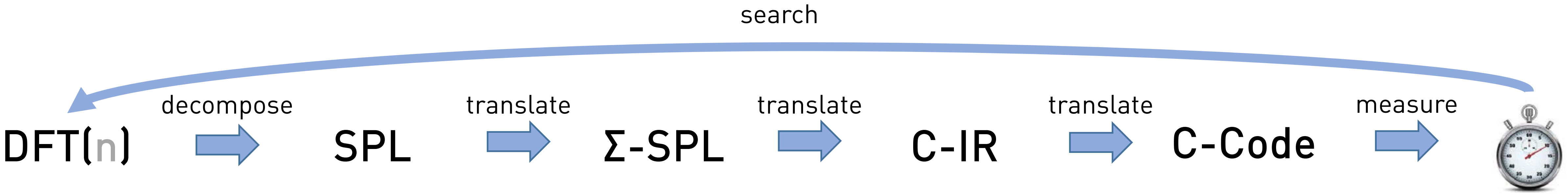
search

DFT(n) decompose SPL translate Σ-SPL translate C-IR translate C-Code measure

Can we utilize modern program language features for:

- DSLs
  - Translation between DSLs
  - Rewrites on DSLs
- Abstraction over
  - Low level transformations
  - Data layouts

My Research



search

DFT(n) decompose SPL translate Σ-SPL translate C-IR translate C-Code measure

Can we utilize modern program language features for:

- DSLs
  - Translation between DSLs
  - Rewrites on DSLs
- Abstraction over
  - Low level transformations
  - Data layouts

Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries
Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky and Markus Püschel (Proc. International Conference on Generative Programming: Concepts & Experiences (GPCE), pp. 125-134, 2013)

Content of this Lecture

DSLs

- Use Case
- External vs. Internal (embedded) DSL
  - Shallow vs. Deep embedding

Staging

- What is Staging?
- Staging in Scala
- Deep DSL Embedding through Staging

Abstracting Meta Programming

- Overview and Motivation
- Abstracting Code Style
- Abstracting Data Layout

DSLs

A language that captures a domain (not general purpose)

- More concise
- Quicker to write
- Easier to maintain
- Easier to reason about

DSLs

Implementation can be roughly categorized into

- External DSLs
- Internal (Embedded) DSLs
  - Shallow Embedding
  - Deep Embedding

External DSLs

External DSLs

Most famous

Older DSLs tend to be external DSLs

Require to design syntax, semantics and tools for the language

Reinventing the wheel for your own language (IR optimizations etc.)

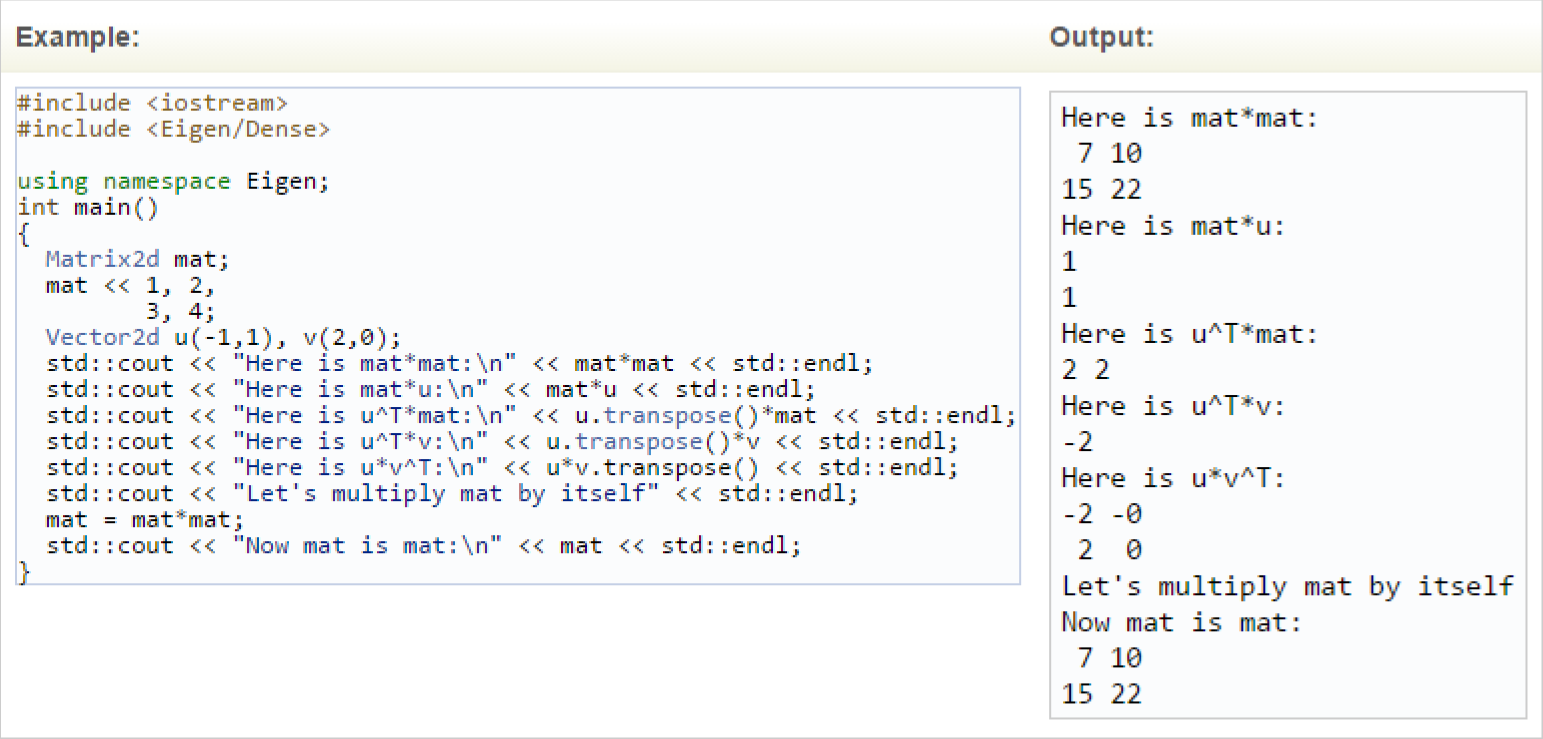
Usually only pays off for “large” efforts

Internal (Embedded DSLs)

- Use (abuse) the parsing infrastructure of the host language
  - Limited by it – easier in modern languages

Internal (Embedded DSLs)

- Use (abuse) the parsing infrastructure of the host language
  - Limited by it – easier in modern languages



Example:

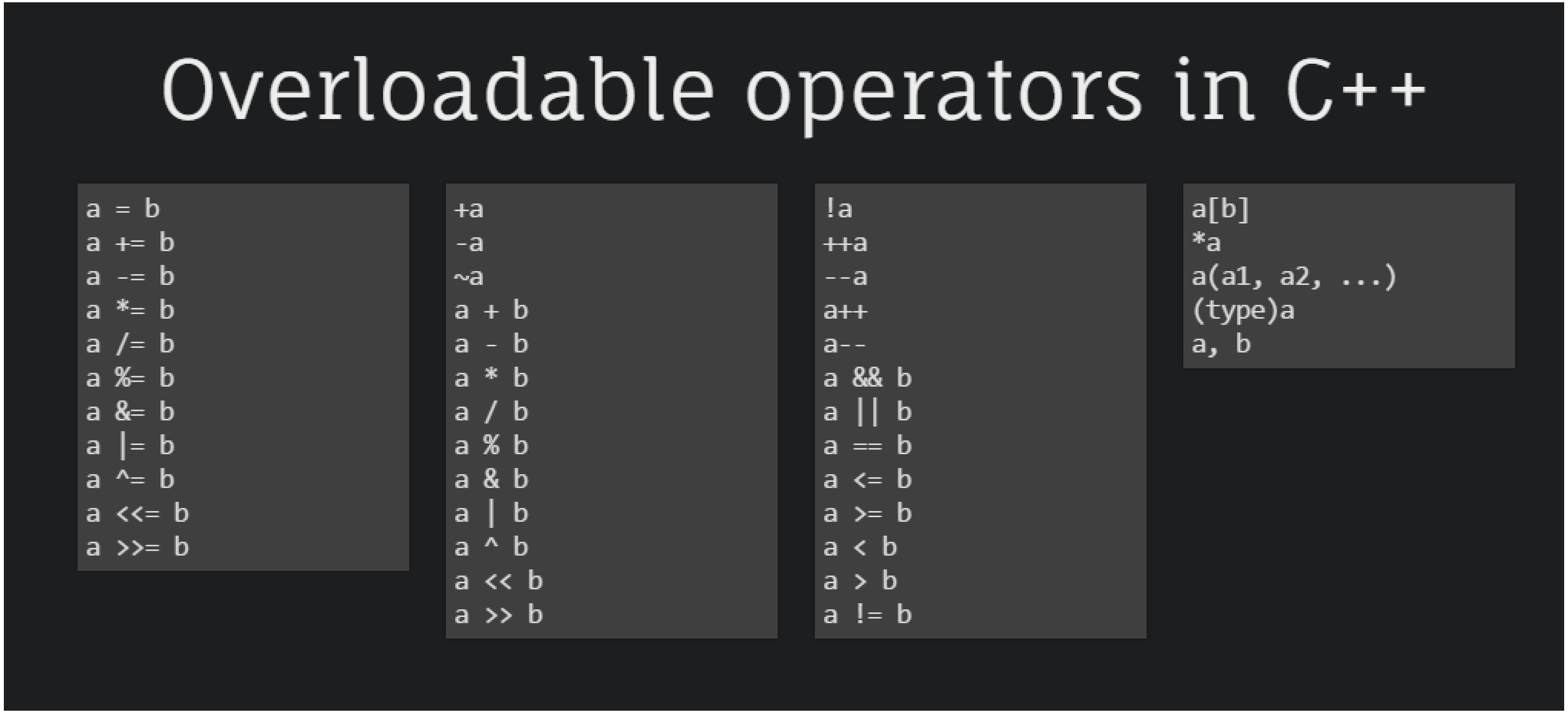
```cpp
#include <iostream>
#include <Eigen/Dense>

using namespace Eigen;
int main()
{
  Matrix2d mat;
  mat << 1, 2,
         3, 4;
  Vector2d u(-1,1), v(2,0);
  std::cout << "Here is mat*mat:\n" << mat*mat << std::endl;
  std::cout << "Here is mat*u:\n" << mat*u << std::endl;
  std::cout << "Here is u^T*mat:\n" << u.transpose()*mat << std::endl;
  std::cout << "Here is u^T*v:\n" << u.transpose()*v << std::endl;
  std::cout << "Here is u*v^T:\n" << u*v.transpose() << std::endl;
  std::cout << "Let's multiply mat by itself" << std::endl;
  mat = mat*mat;
  std::cout << "Now mat is mat:\n" << mat << std::endl;
}
```

Output:

```
Here is mat*mat:
 7 10
15 22
Here is mat*u:
1
1
Here is u^T*mat:
2 2
Here is u^T*v:
-2
Here is u*v^T:
-2 -0
 2  0
Let's multiply mat by itself
Now mat is mat:
 7 10
15 22
```

Internal (Embedded DSLs)

- Use (abuse) the parsing infrastructure of the host language
  - Limited by it – easier in modern languages



Overloadable operators in C++

```
a = b
a += b
a -= b
a *= b
a /= b
a %= b
a &= b
a |= b
a ^= b
a <<= b
a >>= b
```

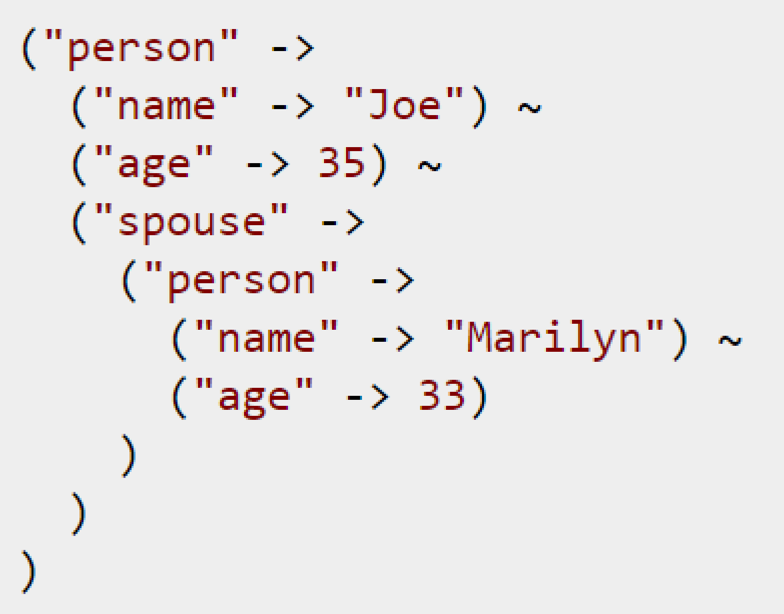
```
+a
-a
~a
a + b
a - b
a * b
a / b
a % b
a & b
a | b
a ^ b
a << b
a >> b
```

```
!a
++a
--a
a++
a--
a && b
a || b
a == b
a <= b
a >= b
a < b
a > b
a != b
```

```
a[b]
*a
a(a1, a2, ...)
(type)a
a, b
```

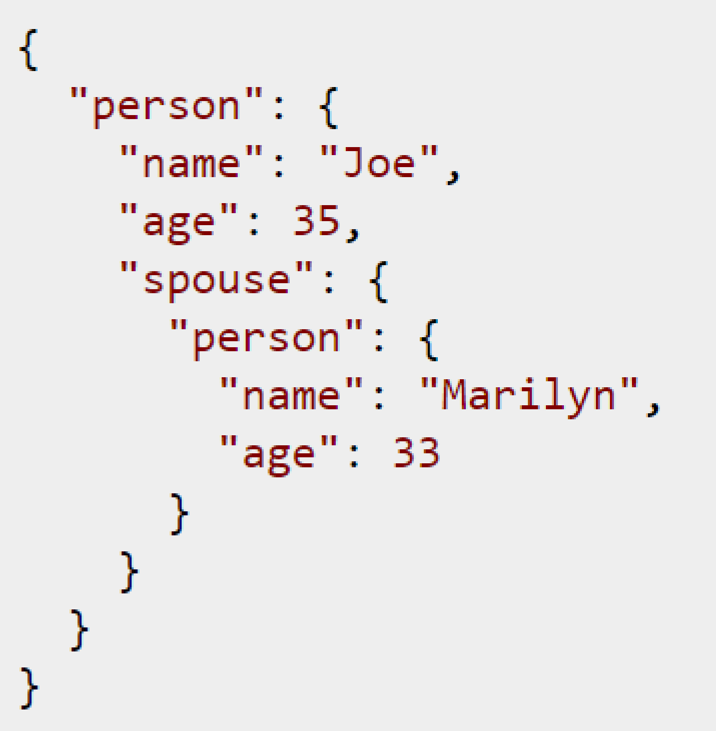
Internal (Embedded DSLs)

- Use (abuse) the parsing infrastructure of the host language
  - Limited by it – easier in modern languages



```
("person" ->
  ("name" -> "Joe") ~
  ("age" -> 35) ~
  ("spouse" ->
    ("person" ->
      ("name" -> "Marilyn") ~
      ("age" -> 33)
    )
  )
)
```

Scala



```
{
  "person": {
    "name": "Joe",
    "age": 35,
    "spouse": {
      "person": {
        "name": "Marilyn",
        "age": 33
      }
    }
  }
}
```

Shallow Embedded DSLs

Shallow Embedded DSLs

Syntactic Sugar for a library

Uses (abuses) the syntax of the host language

No external tools required

easy to deploy

whole program optimization very hard

debugging potentially painful

(e.g. templates)

Deep Embedded DSLs

Delite

Feldspar

Deeply Embedded DSLs

Creates an IR instead of calling a library

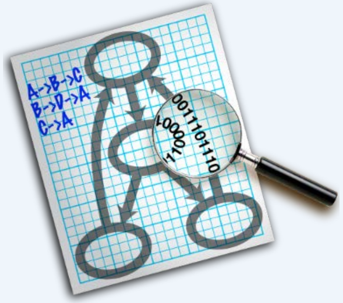
expose IR manipulation to user

Uses (abuses) the syntax of the host language

Infrastructure for handling the IR required

no broad adaptation yet

DSLs



MATLAB SIMULINK

SQL

GNU MAKE

HTML CSS

powered by Lua

LaTeX

OpenGL

External DSLs

hadoop

boost C++ LIBRARIES

Delite

Σ-SPL

Spark

Feldspar

Embedded DSLs

Content of this Lecture

DSLs

- Use Case
- External vs. Internal (embedded) DSL
  - Shallow vs. Deep embedding

Staging

- What is Staging?
- Staging in Scala
- Deep DSL Embedding through Staging

Abstracting Meta Programming

- Overview and Motivation
- Abstracting Code Style
- Abstracting Data Layout

Staging

```
def generic_power(b: Double, n: Int): Double = {if(n == 0) 1.0 else b * generic_power(b,n -1) }
```

Staging

```
def generic_power(b: Double, n: Int): Double = {if(n == 0) 1.0 else b * generic_power(b,n -1) }

def specialized_power3(b: Double): Double = { b * b * b }
```

Staging

```
def generic_power(b: Double, n: Int): Double = {if(n == 0) 1.0 else b * generic_power(b,n -1) }

def specialized_power3(b: Double): Double = { b * b * b }

def specialized_power4(b: Double): Double = { b * b * b * b }
```

Staging

```
def generic_power(b: Double, n: Int): Double = {if(n == 0) 1.0 else b * generic_power(b,n -1) }

def specialized_power3(b: Double): Double = { b * b * b }

def specialized_power4(b: Double): Double = { b * b * b * b }

def specialized_power5(b: Double): Double = { b * b * b * b * b }
```

Staging

```
def generic_power(b: Double, n: Int): Double = {if(n == 0) 1.0 else b * generic_power(b,n -1) }

def specialized_power3(b: Double): Double = { b * b * b }

def specialized_power4(b: Double): Double = { b * b * b * b }

def specialized_power5(b: Double): Double = { b * b * b * b * b }

def specialized_power_generator(n: Int): String ={
    def recurse(n: Int): String = if(n == 1) " b" else "b * " + recurse(n-1)
    "def specialized_power"+n+"(b: Double):Double ={ " + recurse(n) + " }"
  }
```

Staging

```
def generic_power(b: Double, n: Int): Double = {if(n == 0) 1.0 else b * generic_power(b,n -1) }

def specialized_power3(b: Double): Double = { b * b * b }

def specialized_power4(b: Double): Double = { b * b * b * b }

def specialized_power5(b: Double): Double = { b * b * b * b * b }

def specialized_power_generator(n: Int): String ={
    def recurse(n: Int): String = if(n == 1) " b" else "b * " + recurse(n-1)
    "def specialized_power"+n+"(b: Double):Double ={ " + recurse(n) + " }"
  }

println(specialized_power_generator(6))
```

Staging

```
def generic_power(b: Double, n: Int): Double = {if(n == 0) 1.0 else b * generic_power(b,n -1) }

def specialized_power3(b: Double): Double = { b * b * b }

def specialized_power4(b: Double): Double = { b * b * b * b }

def specialized_power5(b: Double): Double = { b * b * b * b * b }

def specialized_power_generator(n: Int): String ={
    def recurse(n: Int): String = if(n == 1) " b" else "b * " + recurse(n-1)
    "def specialized_power"+n+"(b: Double):Double ={ " + recurse(n) + " }"
  }

println(specialized_power_generator(6))

        def specialized_power6(b: Double):Double ={ b * b * b * b * b *  b }
```

Staging

```
def specialized_power_generator(n: Int): String ={
  def recurse(n: Int): String = if(n == 1) " b" else "b * " + recurse(n-1)
  "def specialized_power"+n+"(b: Double):Double ={ " + recurse(n) + " }"
 }
```

Staging

```
def specialized_power_generator(n: Int): String ={
    def recurse(n: Int): String = if(n == 1) " b" else "b * " + recurse(n-1)
    "def specialized_power"+n+"(b: Double):Double ={ " + recurse(n) + " }"
  }
```

Target Code (next Stage)

Staging

Meta Code (current Stage)

```
def specialized_power_generator(n: Int): String ={
    def recurse(n: Int): String = if(n == 1) " b" else "b * " + recurse(n-1)
    "def specialized_power"+n+"(b: Double):Double ={ " + recurse(n) + " }"
  }
```

Meta Code (current Stage)

Staging

```
#ifdef OPTION
               y[0] *= OPTION
#else
               y[0] *= 0
#endif
```

Staging

```
#ifdef OPTION
               y[0] *= OPTION
#else
               y[0] *= 0
#endif
```

Target Code (next Stage)

Staging

Meta Code (current Stage)

```
#ifdef OPTION
               y[0] *= OPTION
#else
               y[0] *= 0
#endif
```

Compiler Assisted Staging

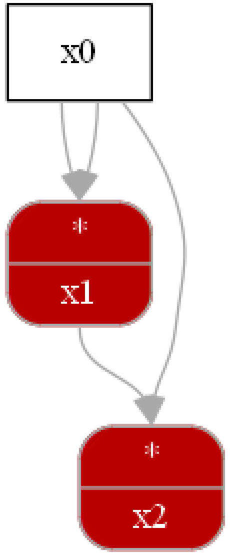
```
case class Rep[T](i: T){
   def * (lhs: Rep[T]): Rep[T] = createGraphNode(this,lhs)
   def createGraphNode(x: Rep[T], y: Rep[T]): Rep[T] = ???
 }
```

```
 def staged_power(b: Rep[Double], n: Int): Rep[Double] = {
   if(n == 1) b else b * staged_power(b,n -1)
 }
```

Compiler Assisted Staging

```
case class Rep[T](i: T){
   def * (lhs: Rep[T]): Rep[T] = createGraphNode(this,lhs)
   def createGraphNode(x: Rep[T], y: Rep[T]): Rep[T] = ???
 }
```

```
 def staged_power(b: Rep[Double], n: Int): Rep[Double] = {
   if(n == 1) b else b * staged_power(b,n -1)
 }
```

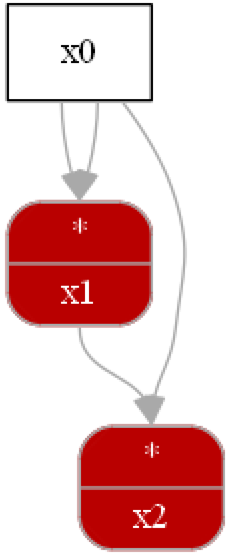


Compiler Assisted Staging

```
case class Rep[T](i: T){
   def * (lhs: Rep[T]): Rep[T] = createGraphNode(this,lhs)
   def createGraphNode(x: Rep[T], y: Rep[T]): Rep[T] = ???
 }
```

```
def staged_power(b: Rep[Double], n: Int): Rep[Double] = {
  if(n == 1) b else b * staged_power(b,n -1)
}
```

Target Code (next Stage)

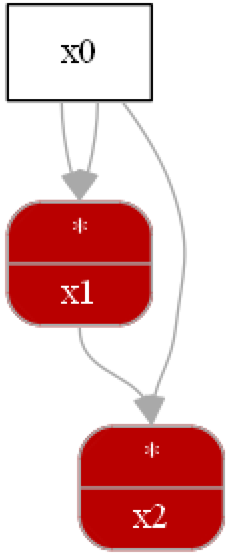


Compiler Assisted Staging

```
case class Rep[T](i: T){
   def * (lhs: Rep[T]): Rep[T] = createGraphNode(this,lhs)
   def createGraphNode(x: Rep[T], y: Rep[T]): Rep[T] = ???
 }
```

Meta Code (current Stage)

```
def staged_power(b: Rep[Double], n: Int): Rep[Double] = {
   if(n == 1) b else b * staged_power(b,n -1)
 }
```



Deep DSL Embedding through Staging

```
abstract class SPL

case class I(n: Int) extends SPL
case class F_2() extends SPL
case class Const(c: SPL) extends Rep[SPL](c)

implicit def SPLtoRep (i: SPL): Rep[SPL] = Const(i)
def tensor(x: Rep[SPL], y: Rep[SPL]): Rep[SPL] = x.createGraphNode(x, y)
def compose(x: Rep[SPL], y: Rep[SPL]): Rep[SPL] = x.createGraphNode(x, y)
```

Deep DSL Embedding through Staging

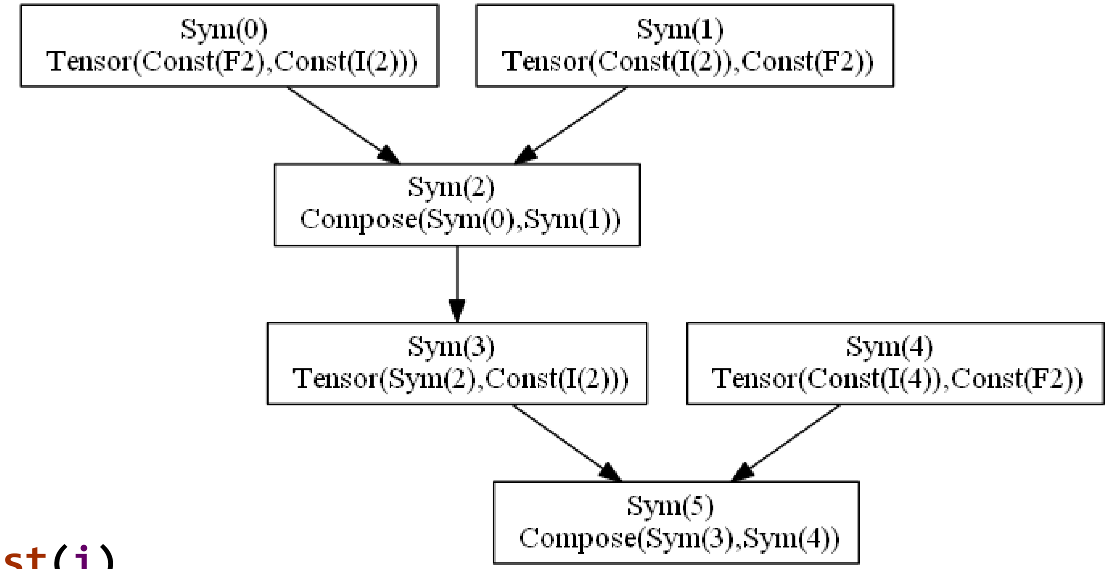
```
abstract class SPL

case class I(n: Int) extends SPL
case class F_2() extends SPL
case class Const(c: SPL) extends Rep[SPL](c)

implicit def SPLtoRep (i: SPL): Rep[SPL] = Const(i)
def tensor(x: Rep[SPL], y: Rep[SPL]): Rep[SPL] = x.createGraphNode(x, y)
def compose(x: Rep[SPL], y: Rep[SPL]): Rep[SPL] = x.createGraphNode(x, y)
```

((F_2() tensor I(2)) compose (I(2) tensor F_2()) tensor I(2)) compose (I(4) tensor F_2())

$$(((F2 \otimes I(2))(I(2) \otimes F2)) \otimes I(2))(I(4) \otimes F2)$$

Deep DSL Embedding through Staging

```
abstract class SPL

case class I(n: Int) extends SPL
case class F_2() extends SPL
case class Const(c: SPL) extends Rep[SPL](c)

implicit def SPLtoRep (i: SPL): Rep[SPL] = Const(i)
def tensor(x: Rep[SPL], y: Rep[SPL]): Rep[SPL] = x.createGraphNode(x, y)
def compose(x: Rep[SPL], y: Rep[SPL]): Rep[SPL] = x.createGraphNode(x, y)
```



Sym(0)
Tensor(Const(F2),Const(I(2)))

Sym(1)
Tensor(Const(I(2)),Const(F2))

Sym(2)
Compose(Sym(0),Sym(1))

Sym(3)
Tensor(Sym(2),Const(I(2)))

Sym(4)
Tensor(Const(I(4)),Const(F2))

Sym(5)
Compose(Sym(3),Sym(4))

((F_2() tensor I(2)) compose (I(2) tensor F_2()) tensor I(2)) compose (I(4) tensor F_2())

$$(((F2 \otimes I(2))(I(2) \otimes F2)) \otimes I(2))(I(4) \otimes F2)$$

Content of this Lecture

DSLs

- Use Case
- External vs. Internal (embedded) DSL
  - Shallow vs. Deep embedding

Staging

- What is Staging?
- Staging in Scala
- Deep DSL Embedding through Staging

Abstracting Meta Programming

- Overview and Motivation
- Abstracting Code Style
- Abstracting Data Layout

Code Style and Data Layouts

Σ-SPL

$$T_m^n \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right)$$

Code Style and Data Layouts

Σ-SPL

Code Style

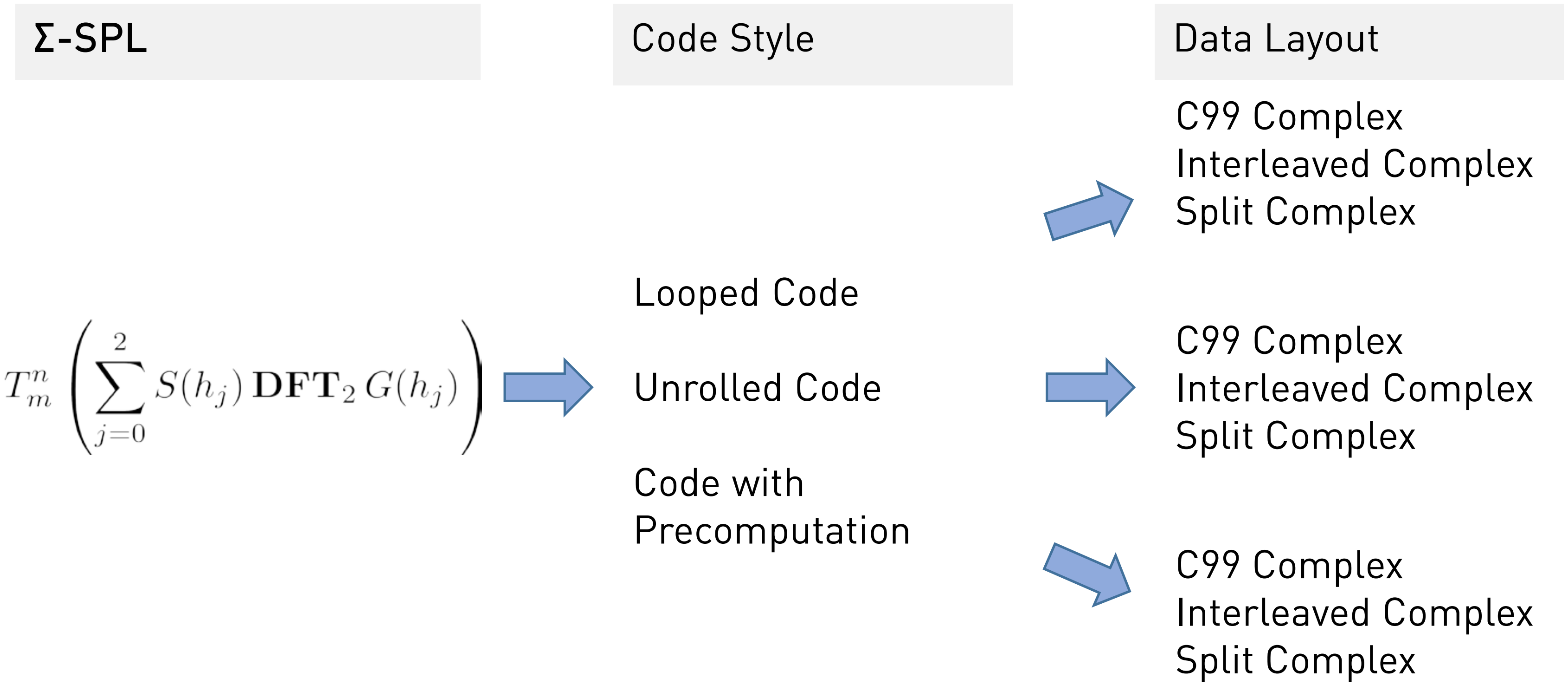
$$T^n_m \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right)$$

Looped Code

Unrolled Code

Code with Precomputation

Code Style and Data Layouts



Σ-SPL

Code Style

Data Layout

$$T^n_m\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

Looped Code

Unrolled Code

Code with Precomputation

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

Code Style and Data Layouts

Σ-SPL

$$T_m^n \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right)$$

Code Style

Looped Code

Unrolled Code

Code with
Precomputation

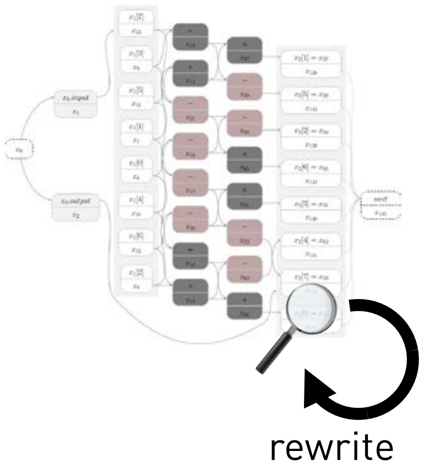
Data Layout

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

AST



Code Style and Data Layouts

Σ-SPL

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

Code Style

Looped Code

Unrolled Code

Code with
Precomputation

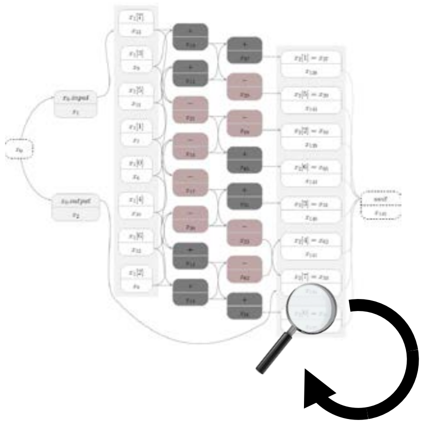
Data Layout

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

AST

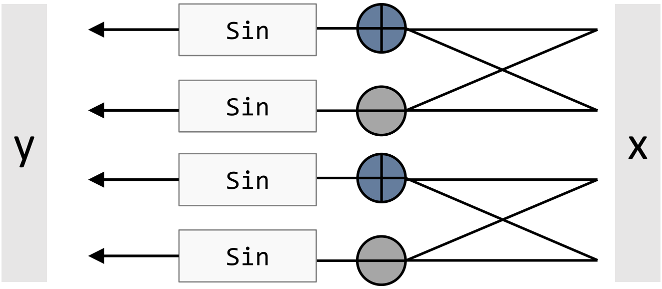


rewrite

Code

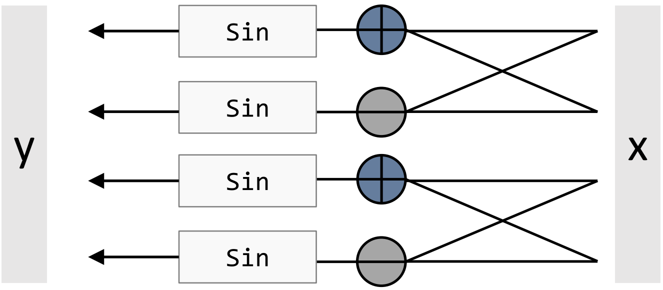
Code Style

$$T_m^n \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right)$$



Code Style

$$T^n_m\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

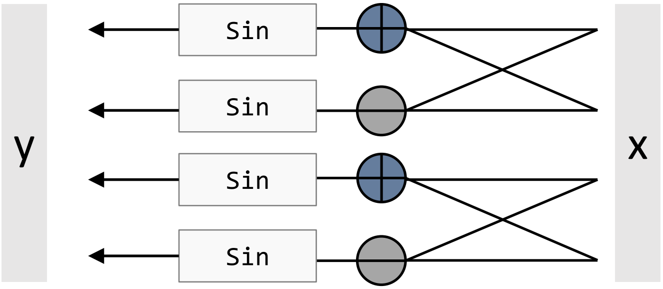


Looped

```
for (int i=0;i < 2;i++) {
   y[i*2]   = x[i*2] + x[i*2+1];
   y[i*2+1] = x[i*2] - x[i*2+1];
}
for (int i=0;i < 4;i++) {
   y[i] = y[i] * sin(PI/4*(i+2));
}
```

Code Style

$$T_m^n \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right)$$



Looped

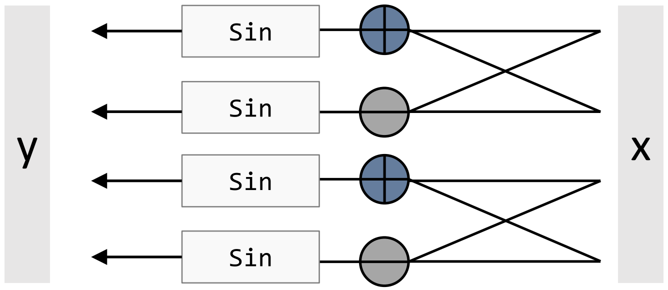
```
for (int i=0;i < 2;i++) {
   y[i*2]   = x[i*2] + x[i*2+1];
   y[i*2+1] = x[i*2] - x[i*2+1];
}
for (int i=0;i < 4;i++) {
   y[i] = y[i] * sin(PI/4*(i+2));
}
```

Unrolled

```
y[0] = x[0] + x[1];
y[1] = x[0] - x[1];
y[2] = x[2] + x[3];
y[3] = x[2] - x[3];

y[0] = y[0] * sin(PI/4*(2));
y[1] = y[1] * sin(PI/4*(3));
y[2] = y[2] * sin(PI/4*(4));
y[3] = y[3] * sin(PI/4*(5));
```

Code Style

$$T^{n}_{m}\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$



Looped

```
for (int i=0;i < 2;i++) {
   y[i*2]   = x[i*2] + x[i*2+1];
   y[i*2+1] = x[i*2] - x[i*2+1];
}
for (int i=0;i < 4;i++) {
   y[i] = y[i] * sin(PI/4*(i+2));
}
```

Unrolled

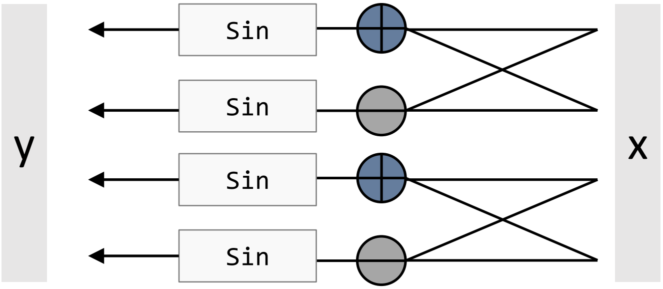
```
y[0] = x[0] + x[1];
y[1] = x[0] - x[1];
y[2] = x[2] + x[3];
y[3] = x[2] - x[3];

y[0] = y[0] * sin(PI/4*(2));
y[1] = y[1] * sin(PI/4*(3));
y[2] = y[2] * sin(PI/4*(4));
y[3] = y[3] * sin(PI/4*(5));
```

Scalarized and Precomputed

```
t0 = x[0] + x[1];
t1 = x[0] - x[1];
t2 = x[2] + x[3];
t3 = x[2] - x[3];

y[0] = t0 * 1;
y[1] = t1 * 0.707;
y[2] = t2 * 0;
y[3] = t3 * -0.707;
```

Data Layout

$$T^n_m \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right)$$

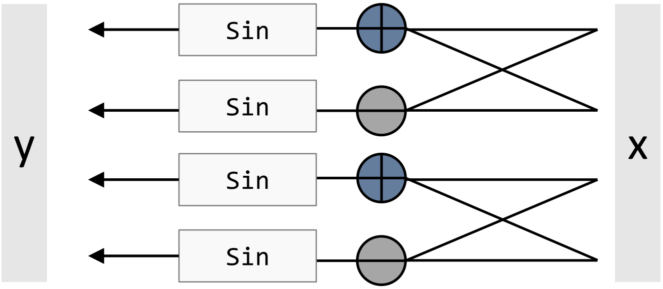


Array of Double

```
for (int i=0;i < 2;i++) {
   y[i*2]   = x[i*2] + x[i*2+1];
   y[i*2+1] = x[i*2] - x[i*2+1];
}
…
```

Data Layout

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$



Array of Double

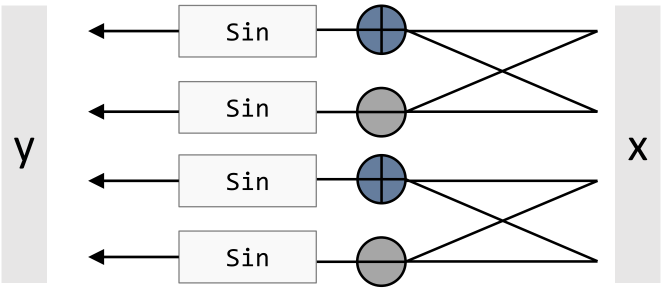
```
for (int i=0;i < 2;i++) {
   y[i*2]   = x[i*2] + x[i*2+1];
   y[i*2+1] = x[i*2] - x[i*2+1];
}
…
```

Interleaved Complex

```
for (int i=0;i < 2;i++) {
   y[i*4]   = x[i*4]   + x[i*4+2]; //real
   y[i*4+1] = x[i*4+1] + x[i*4+3]; //complex

   y[i*4+2] = x[i*4]   - x[i*4+2]; //real
   y[i*4+3] = x[i*4+1] - x[i*4+3]; //complex
}
…
```

Data Layout

$$T^n_m \left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\, G(h_j) \right)$$



Array of Double

```
for (int i=0;i < 2;i++) {
   y[i*2]   = x[i*2] + x[i*2+1];
   y[i*2+1] = x[i*2] - x[i*2+1];
}
…
```

Interleaved Complex

```
for (int i=0;i < 2;i++) {
   y[i*4]   = x[i*4]   + x[i*4+2]; //real
   y[i*4+1] = x[i*4+1] + x[i*4+3]; //complex

   y[i*4+2] = x[i*4]   - x[i*4+2]; //real
   y[i*4+3] = x[i*4+1] - x[i*4+3]; //complex
}
…
```

Split Complex

```
for (int i=0;i < 2;i++) {
   ry[i*2]   = rx[i*2] + rx[i*2+1];
   ry[i*2+1] = rx[i*2] - rx[i*2+1];

   iy[i*2]   = ix[i*2] + ix[i*2+1];
   iy[i*2+1] = ix[i*2] - ix[i*2+1];
}
…
```

Code Style and Data Layouts

Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

Looped Code

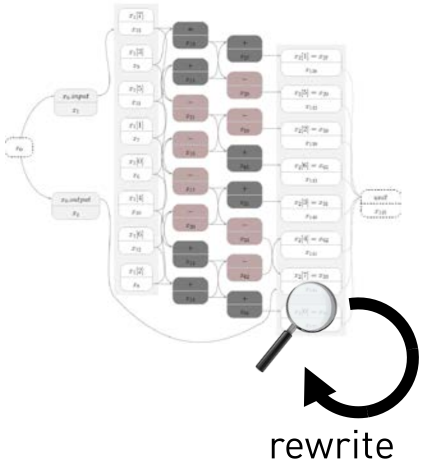
Unrolled Code

Code with Precomputation

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex



Code Style and Data Layouts

Σ-SPL

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

Code Style

Looped Code

Unrolled Code

Code with Precomputation

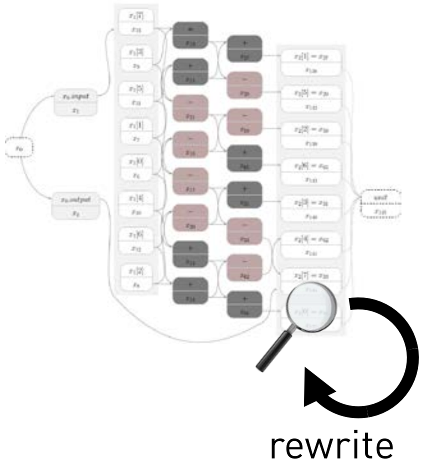
Data Layout

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

AST

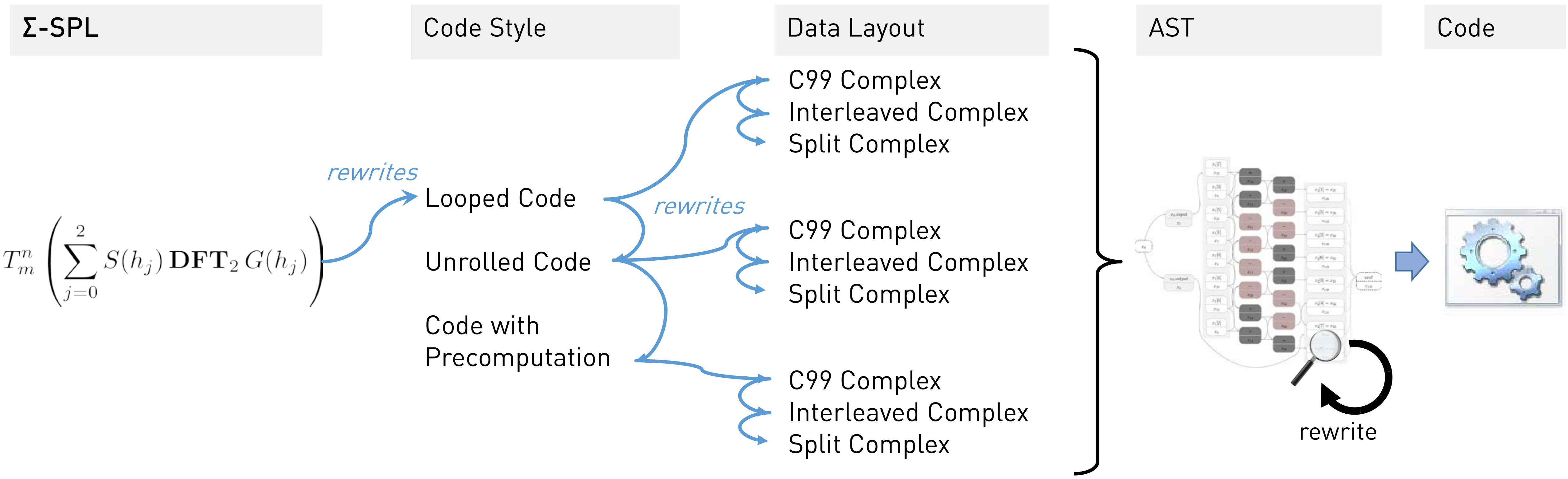


Code

How to build this?

Option 1: “Do it yourself”
Option 2: C++ Meta programming
Option 3: Staging in Scala
…

Option 1: “Do it yourself”



Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

rewrites

Looped Code

Unrolled Code

Code with Precomputation

rewrites

C99 Complex

Interleaved Complex

Split Complex

C99 Complex

Interleaved Complex

Split Complex

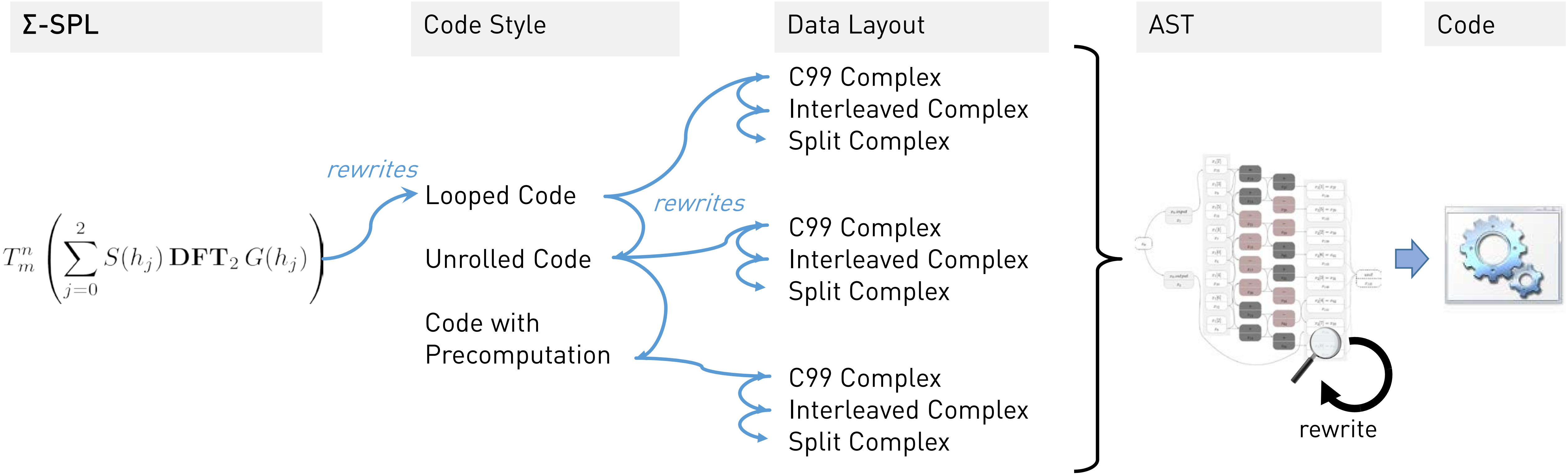
C99 Complex

Interleaved Complex

Split Complex

rewrite

Option 1: “Do it yourself”



Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

rewrites

Looped Code

Unrolled Code

Code with Precomputation

rewrites

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

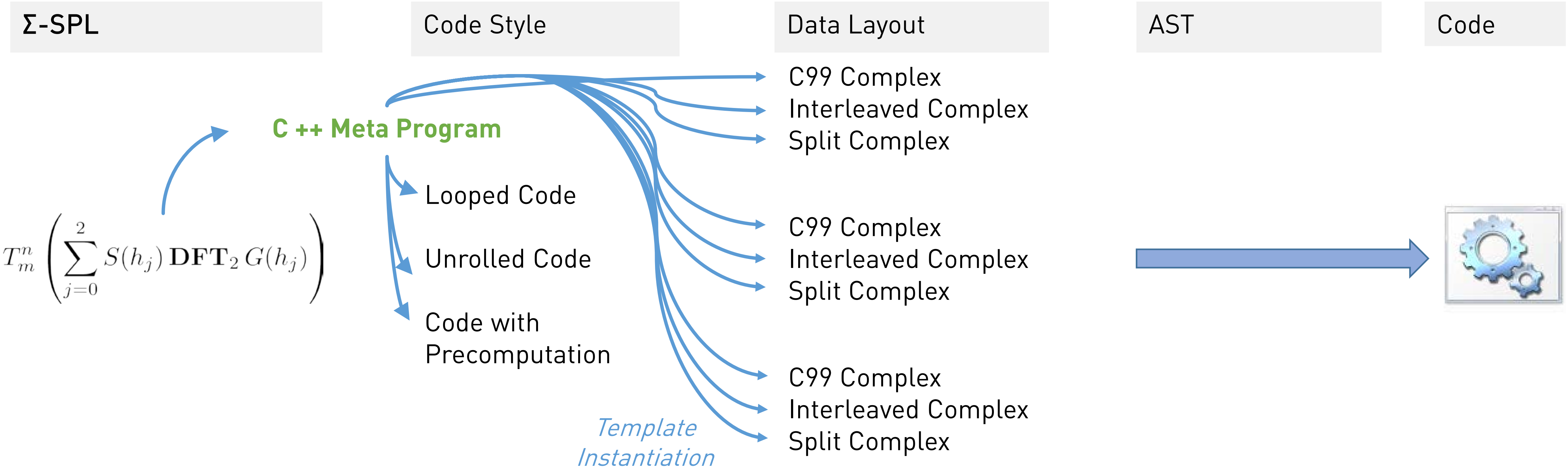
rewrite

Rewrite Engine from Scratch

Large Implementation and Maintenance Effort

Little Reuse between “Backends”

Option 2: C++ Meta Programming



Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

C ++ Meta Program

Looped Code

Unrolled Code

Code with Precomputation

Template Instantiation

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C++ Template Meta Programming

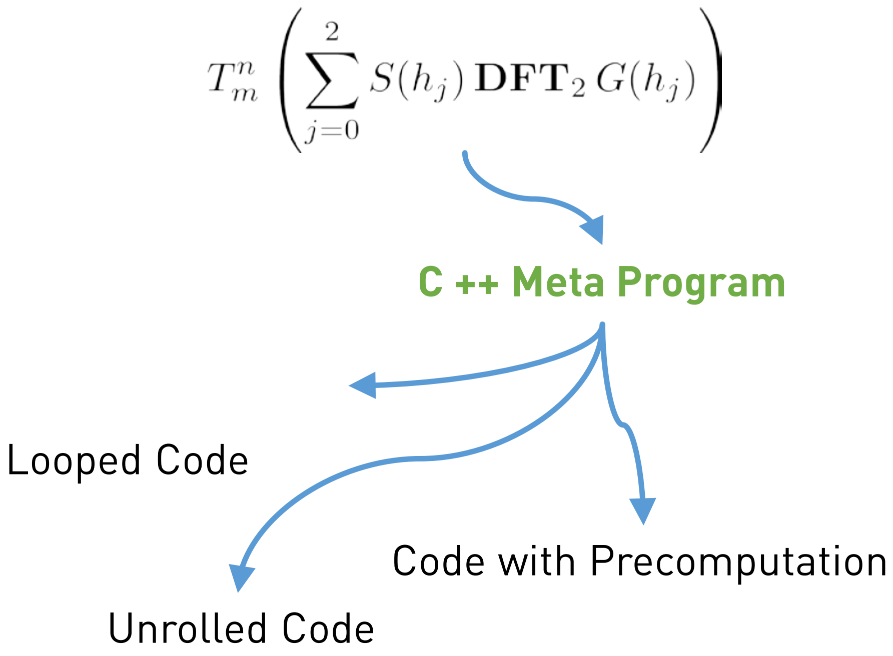
```
template <int T>
struct Fibonacci{    enum { value = (Fibonacci<T - 1>::value +    Fibonacci<T - 2>::value) }; };

template <>
struct Fibonacci<0>{    enum { value = 0 }; };

template <>
struct Fibonacci<1>{    enum { value = 1 }; };

template <>
struct Fibonacci<2>{    enum { value = 1 }; };
```

```
int main()
{
    int x = Fibonacci<45>::value;
    cout << x << endl;
}
```

Template Instantiation

```
int main()
{
    int x = 701408733;
    cout << x << endl;
}
```

C++ Template Meta Programming



$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

C ++ Meta Program

Looped Code

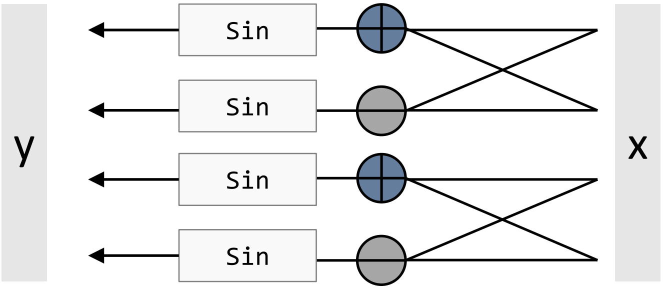
Code with Precomputation

Unrolled Code

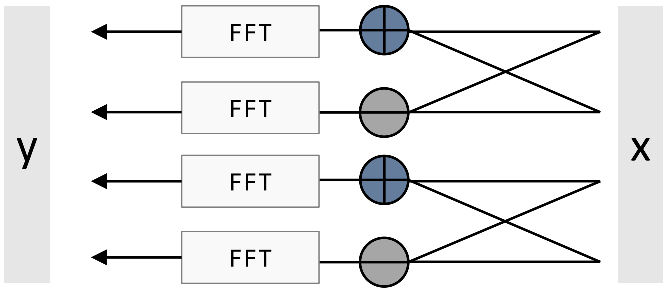
```
auto body1 = [&](const int& i) {
  y[i*2] =   x[i*2] + x[i*2+1];
  y[i*2+1] = x[i*2] - x[i*2+1];
};
for(int i = 0; i < 2; i+= UnrollFact) {
  unroller( body1, i, uint_<UnrollFact - 1 >());
}
auto body2 = [&](const int& i) {
#ifdef PRECOMPUTE
  y[0] *= sin_ct(M_PI/4*(i+2));
#else
  y[0] *= sin(M_PI/4*(i+2));
#endif
};
for(int i = 0; i < 4; i+= UnrollFact) {
  unroller( body2, i, uint_<UnrollFact - 1>() );
}
```

<Templates not shown>

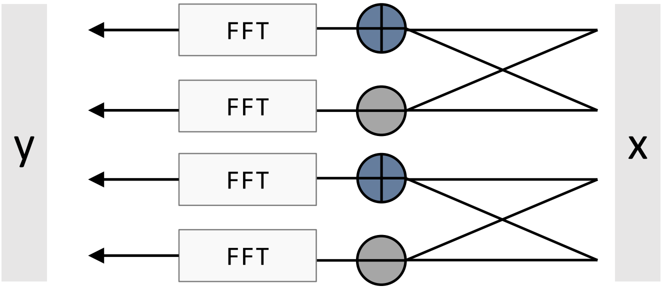
C++ Template Meta Programming



C++ Template Meta Programming

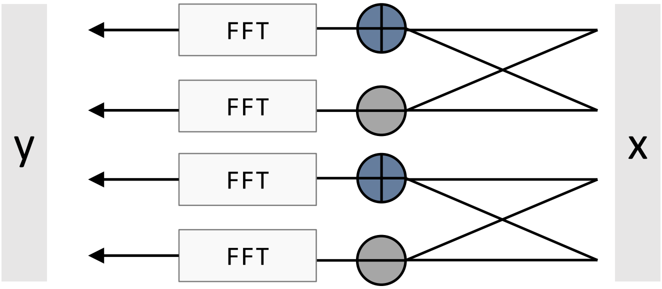


C++ Template Meta Programming



https://github.com/pkeir/ctfft

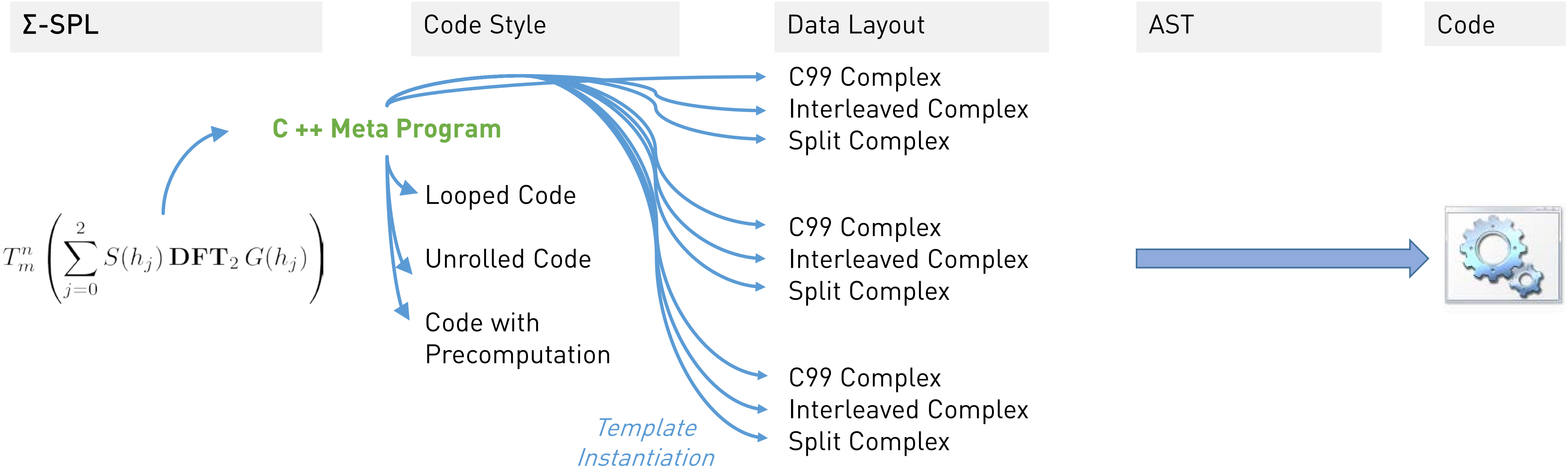
C++ Template Meta Programming



```
cfft.cpp:25:60:   in constexpr expansion of ‘tupconst::map(F, std::tuple<_Elements ...>) [with F =
Comp<cxns::cx<double> (*)(const cxns::cx<double>&), Comp<Init<cxns::cx<double> >,
StaticCast<cxns::cx<double>, long unsigned int> > >; Ts = {long unsigned int, long unsigned int, long
unsigned int, long unsigned int, long unsigned int, long unsigned int, long unsigned int, long unsigned
int}; decltype (tupconst::map_helper(f, t, mk_index_range<0ul, (sizeof (Ts ...) - 1)>())) =
std::tuple<cxns::cx<double>, cxns::cx<double>, cxns::cx<double>, cxns::cx<double>, cxns::cx<double>,
cxns::cx<double>, cxns::cx<double>, cxns::cx<double> >; mk_index_range<0ul, (sizeof (Ts ...) - 1)> =
indicesT<long unsigned int, 0ul, 1ul, 2ul, 3ul, 4ul, 5ul, 6ul, 7ul>](tupconst::iota() [with long
unsigned int N = 8ul; decltype (tupconst::iota_helper(mk_index_range<0ul, (N - 1)>())) = std::tuple<long
unsigned int, long unsigned int, long unsigned int, long unsigned int, long unsigned int, long unsigned
int, long unsigned int, long unsigned int>; mk_index_range<0ul, (N - 1)> = indicesT<long unsigned int,
0ul, 1ul, 2ul, 3ul, 4ul, 5ul, 6ul, 7ul>]())’
```

https://github.com/pkeir/ctfft

Option 2: C++ Meta Programming



Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

C ++ Meta Program

Looped Code

Unrolled Code

Code with Precomputation

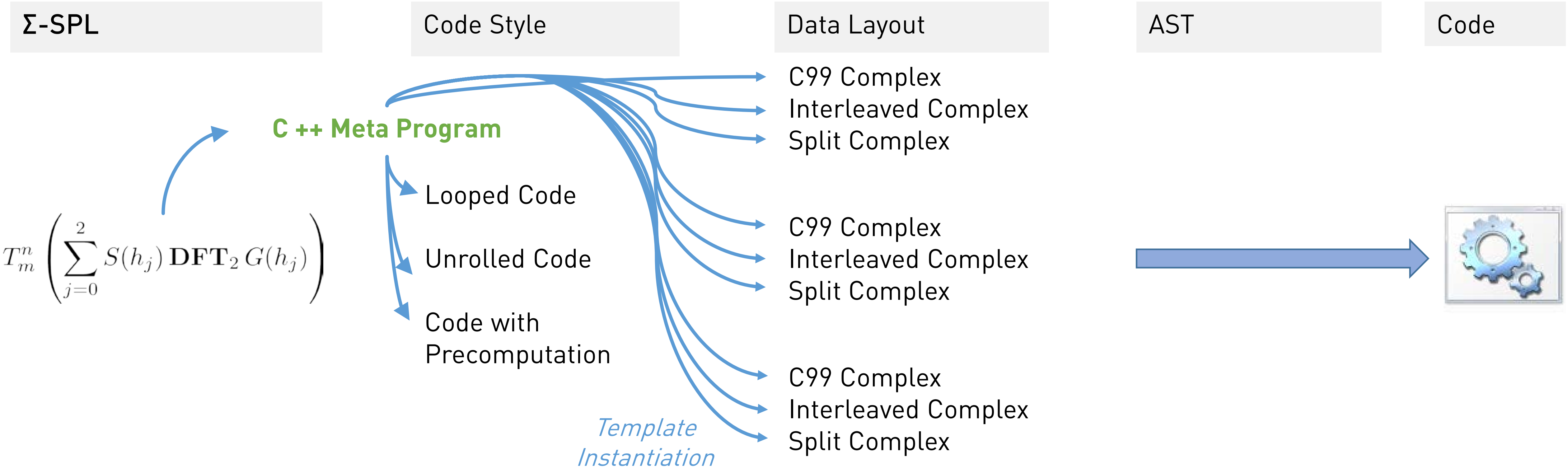
C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

Template Instantiation

Option 2: C++ Meta Programming



Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

C ++ Meta Program

Looped Code

Unrolled Code

Code with Precomputation

Template Instantiation

C99 Complex

Interleaved Complex

Split Complex

C99 Complex

Interleaved Complex

Split Complex

C99 Complex

Interleaved Complex

Split Complex

C++ Meta Programming

One meta program for all targets

Debugging very difficult

Errors only appear when instantiated

By default no explicit AST

Option 3: Staging in Scala

Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right)$$

Scala Staging Program

Looped Code

Unrolled Code

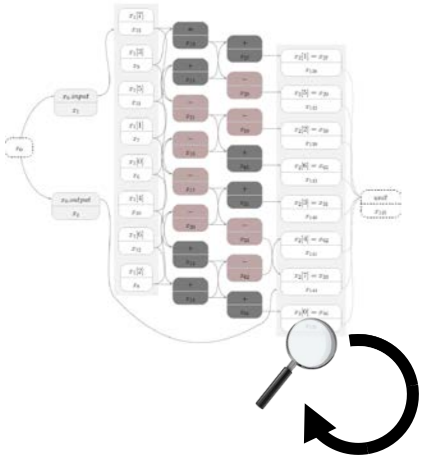
Code with Precomputation

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

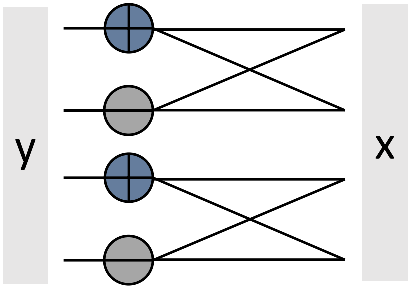
Staging decisions



rewrite

Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```



x				
y				

Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

x				
y				

Staging Applied to the Running Example

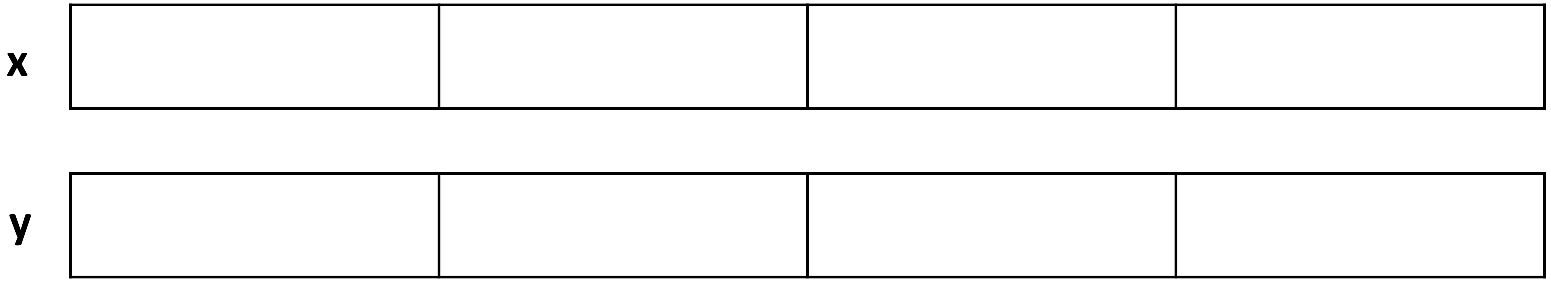
```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

Staging Applied to the Running Example

i = 0

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```



Staging Applied to the Running Example

i = 0

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

$\underbrace{\texttt{x(i*2)}}_{\text{Array.apply(i: Int): T}}$

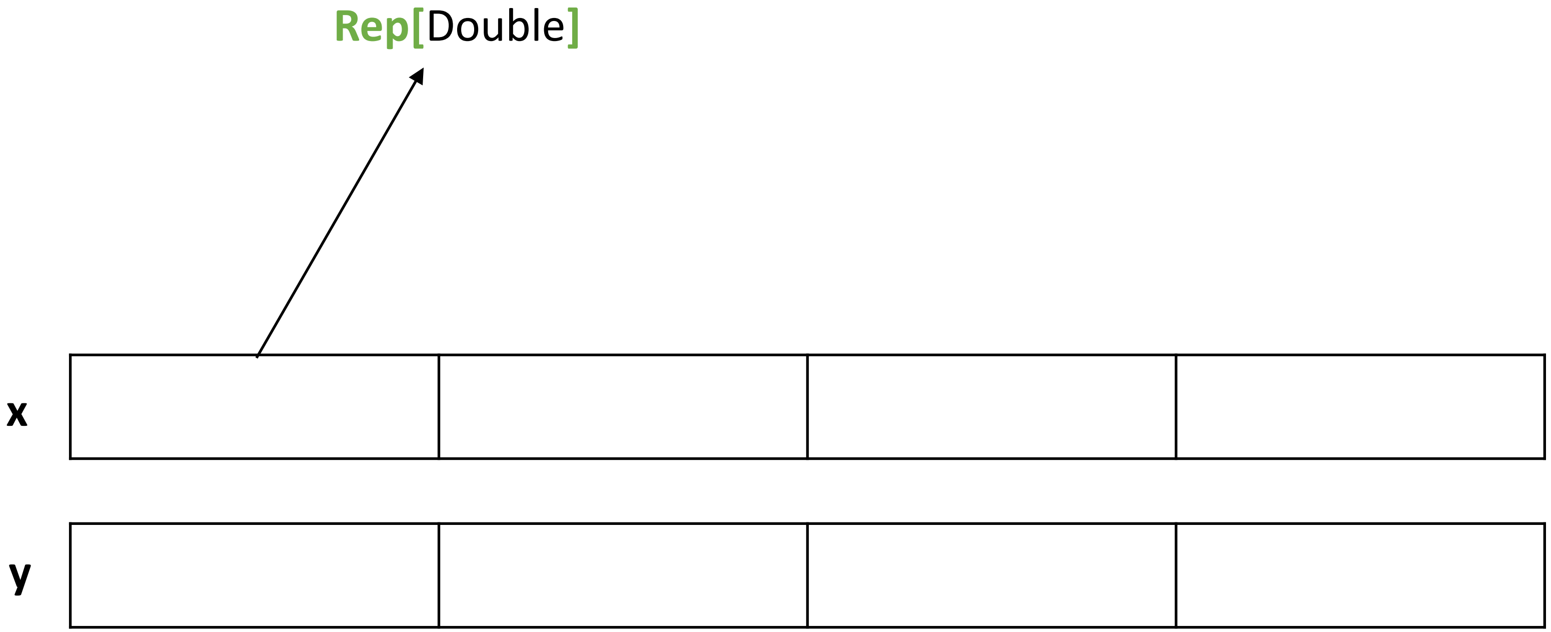
x				
y				

Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

Array.apply(i: Int): T

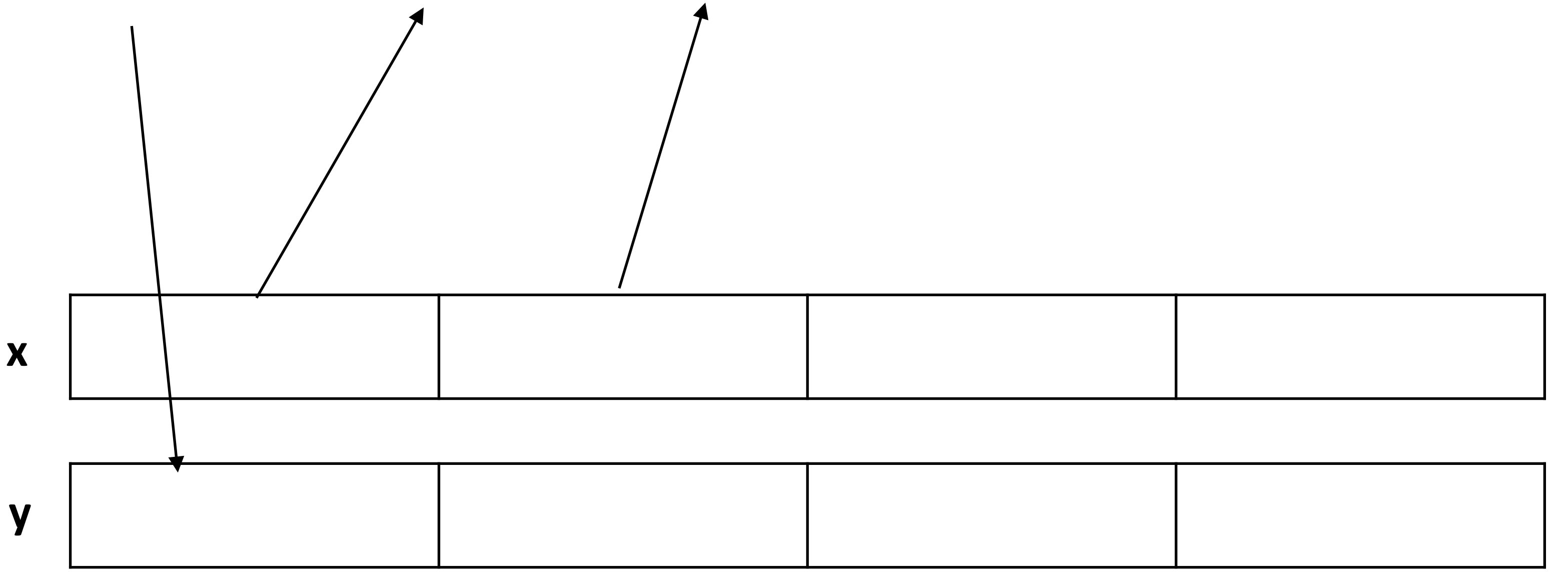


Staging Applied to the Running Example

i = 0

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

Rep[Double] = Rep[Double] + Rep[Double]



Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

$$\texttt{Rep[Double]} = \underbrace{\texttt{Rep[Double]} + \texttt{Rep[Double]}}$$

Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

$$\text{Rep[Double]} = \underbrace{\text{Rep[Double]} + \text{Rep[Double]}}_{\text{How to add two “Reps”?}}$$

Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

$$\text{Rep[Double]} = \underbrace{\text{Rep[Double]} + \text{Rep[Double]}}_{\text{How to add two “Reps”?}}$$

Implementation in Lightweight Modular Staging Library

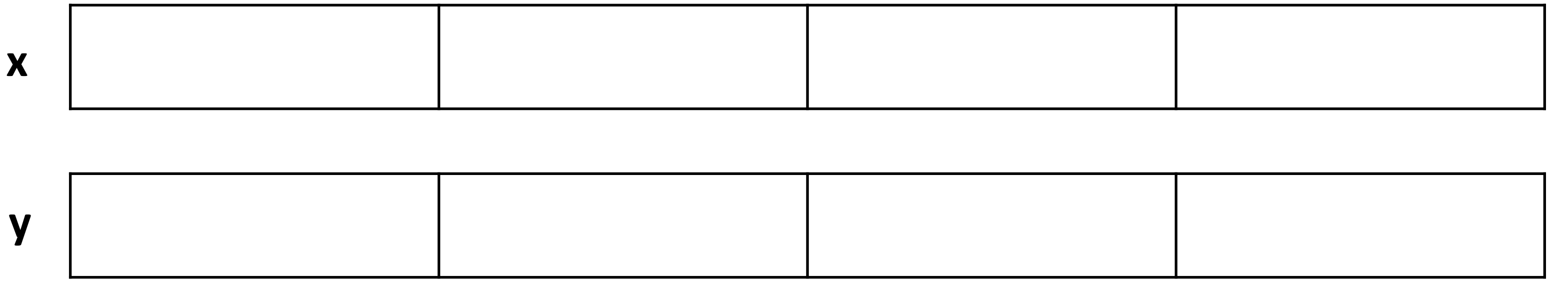
x				
y				

Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

$$\text{Rep[Double]} = \underbrace{\text{Rep[Double]} + \text{Rep[Double]}}_{\text{How to add two “Reps”?}}$$

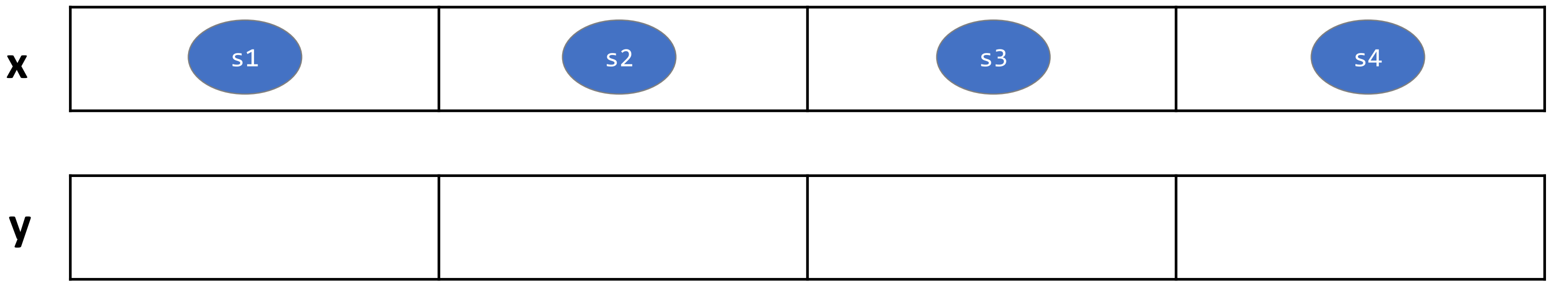


Staging Applied to the Running Example

i = 0

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

$$\text{Rep[Double]} = \underbrace{\text{Rep[Double]} + \text{Rep[Double]}}_{\text{How to add two “Reps”?}}$$

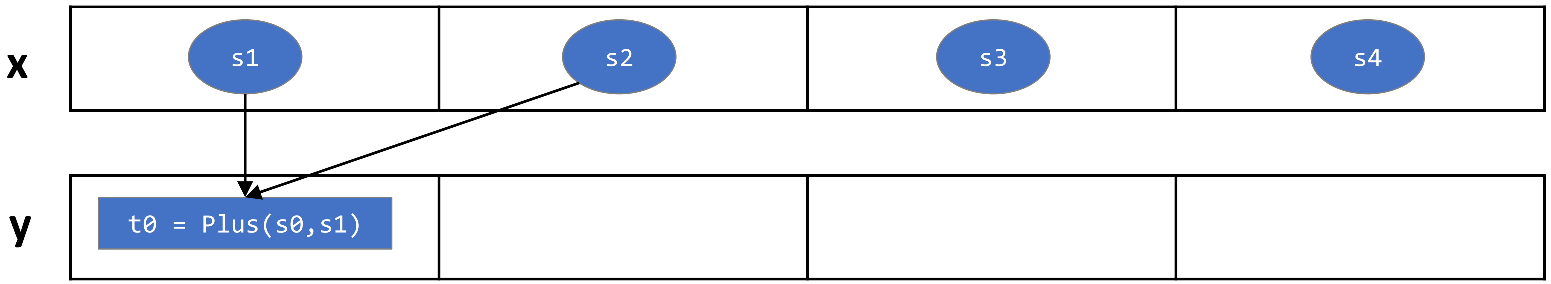


Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

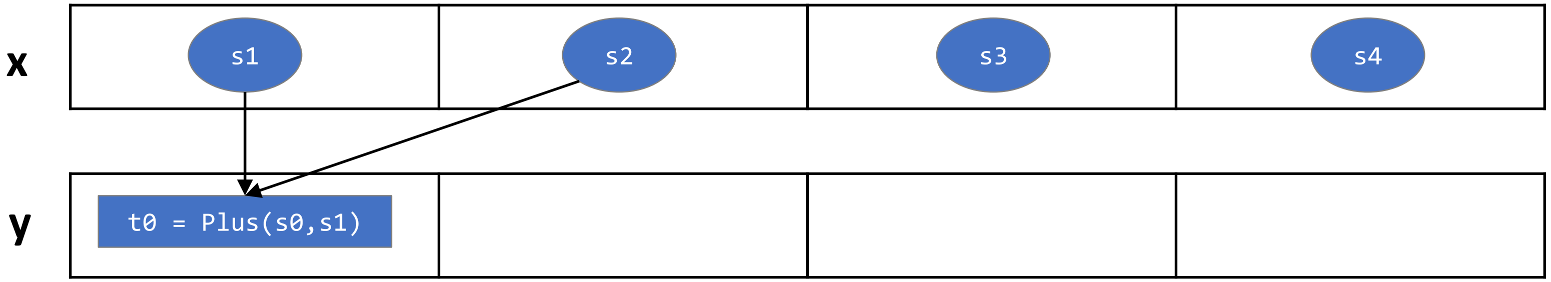
$$\text{Rep[Double]} = \underbrace{\text{Rep[Double]} + \text{Rep[Double]}}_{\text{How to add two “Reps”?}}$$



Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

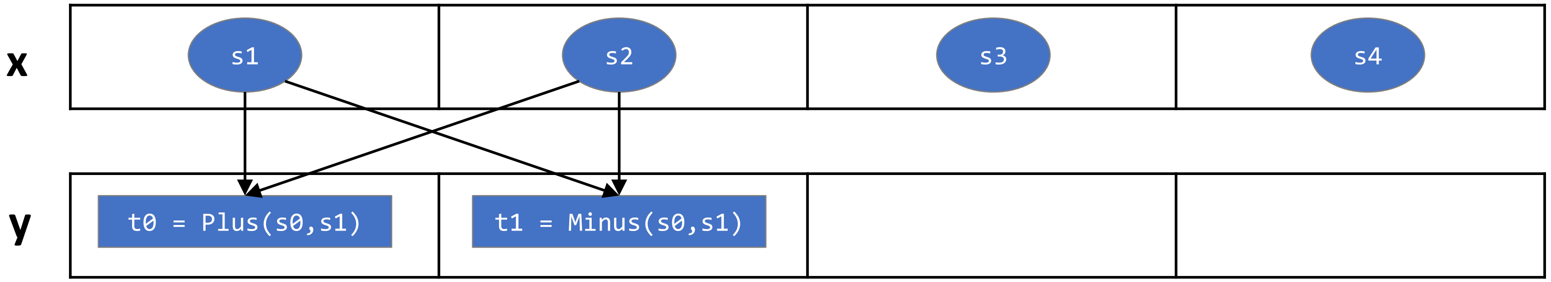
i = 0



Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

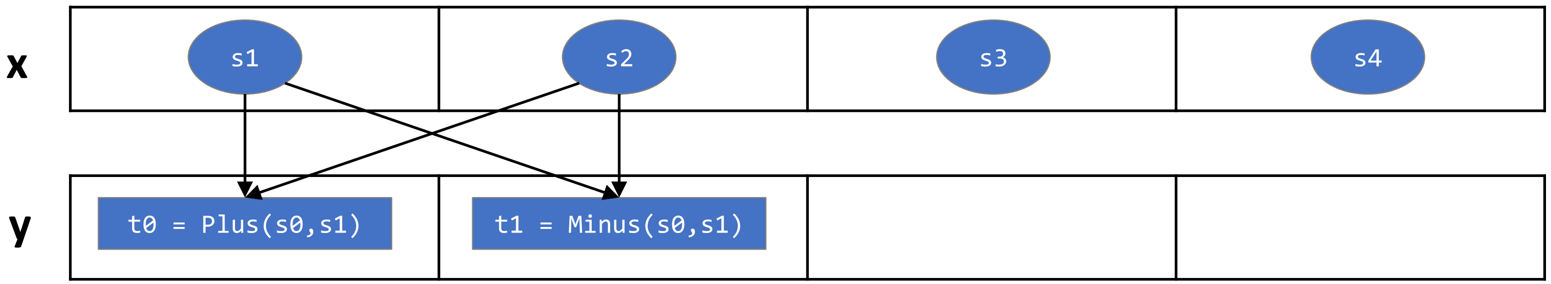
i = 0



Staging Applied to the Running Example

i = 1

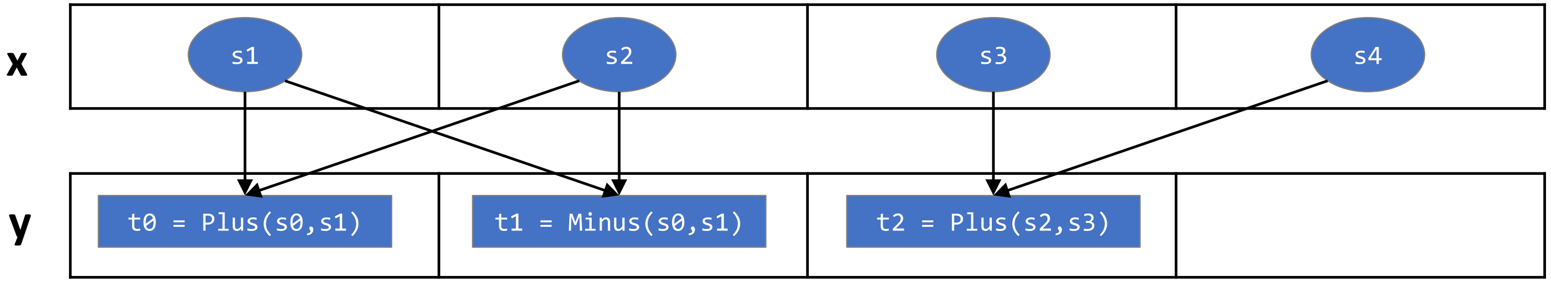
```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
   for (i <- 0 until 2) {
    y(2*i)   = x(i*2) + x(i*2+1)
    y(2*i+1) = x(i*2) - x(i*2+1)
   }
  }
```



Staging Applied to the Running Example

i = 1

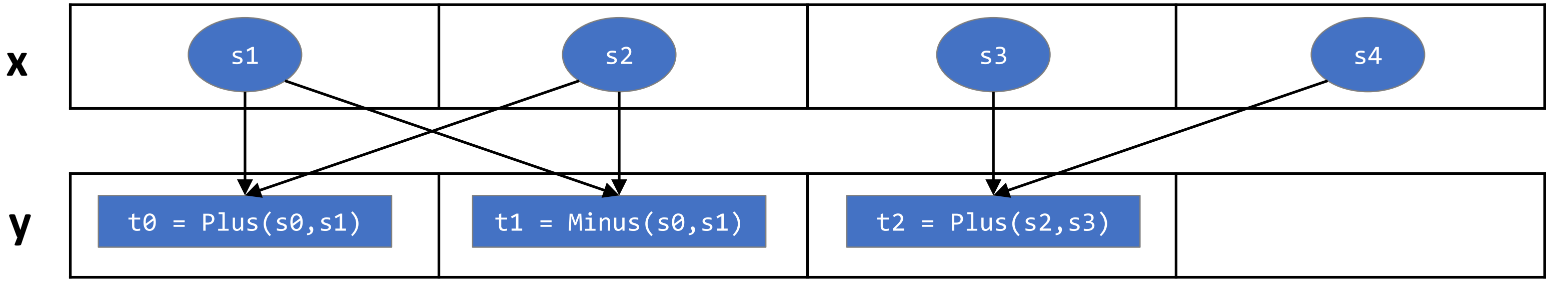
```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```



Staging Applied to the Running Example

i = 1

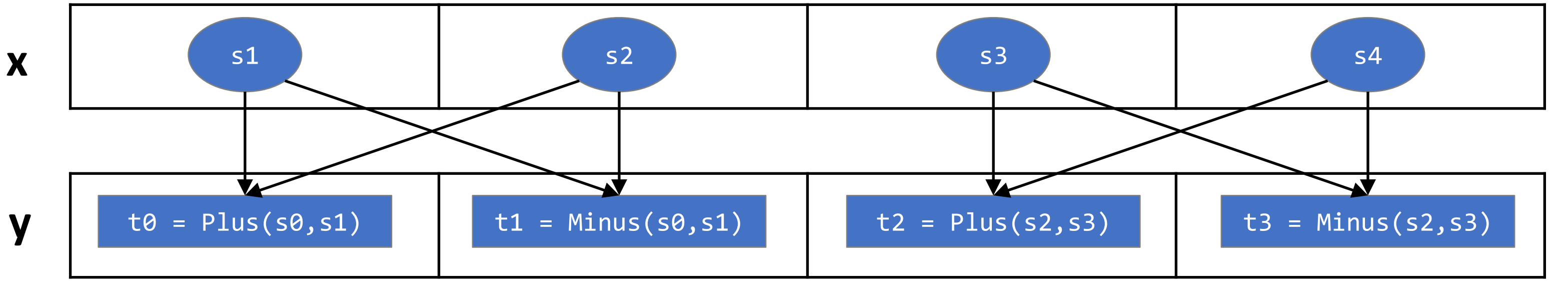
```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```



Staging Applied to the Running Example

i = 1

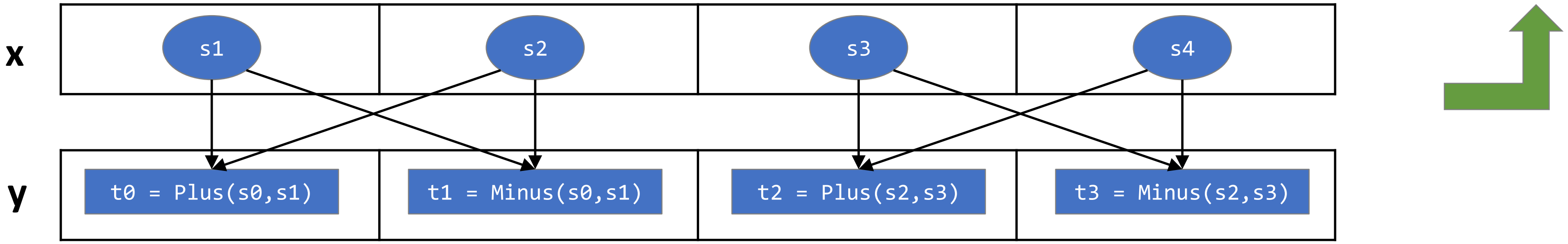
```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
  }
```



Staging Applied to the Running Example

i = 1

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
  }
```

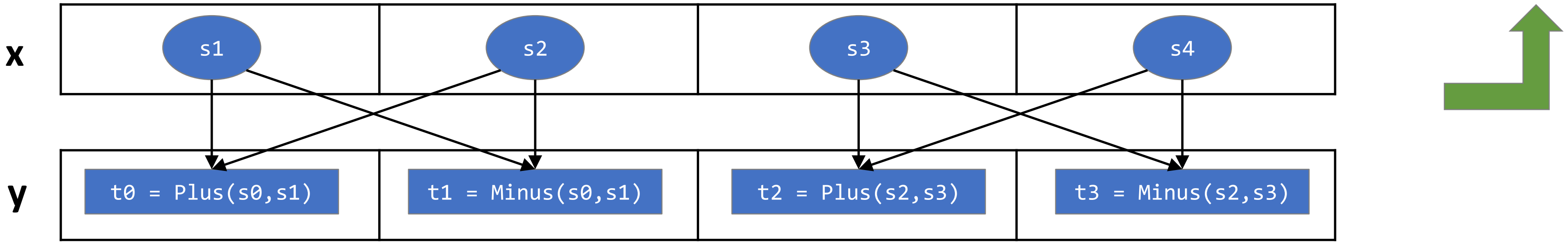


Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

i = 1

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```



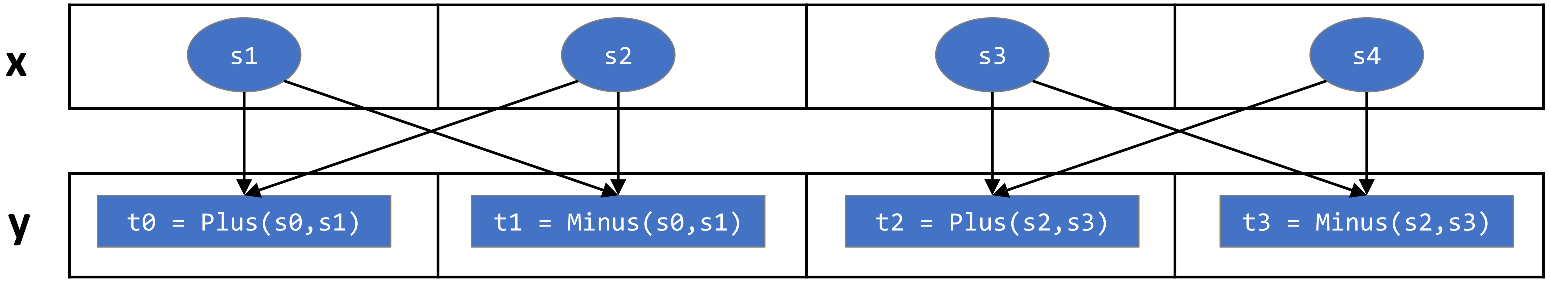
Staging Applied to the Running Example

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

i = 1

Scalars only
Unrolled

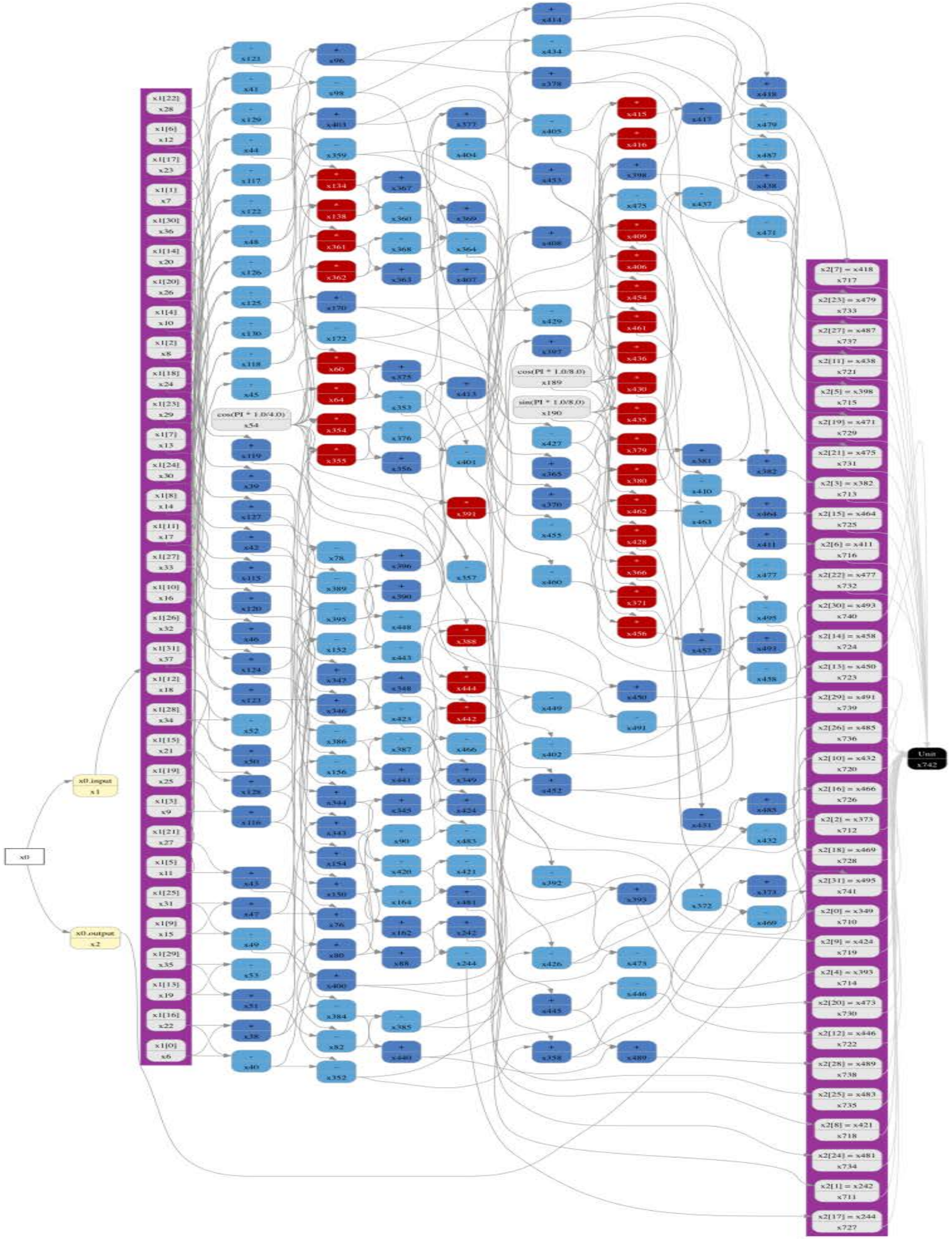
```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

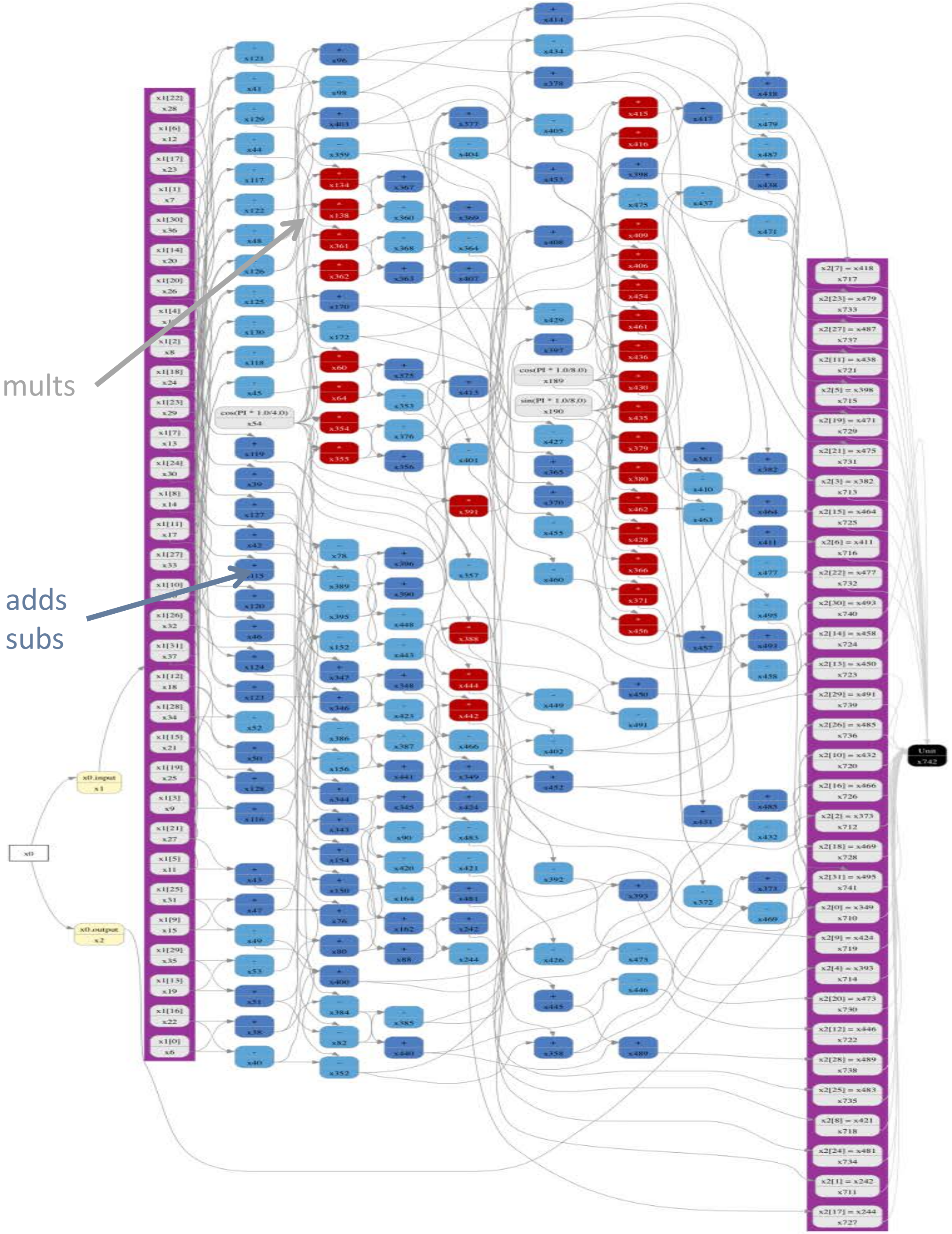


```
trait FFT { this: Arith with Trig =>
  def omega(k: Int, N: Int): Complex = {
    val kth = -2.0 * k * math.Pi / N Complex(cos(kth), sin(kth)) }
  case class Complex(re: Double, im: Double) {
    def +(that: Complex) = Complex(this.re + that.re, this.im + that.im)
    def -(that: Complex) = Complex(this.re - that.re, this.im - that.im)
    def *(that: Complex) = Complex(this.re * that.re - this.im * that.im,
                                 this.re * that.im + this.im * that.re)
  }
  def splitEvenOdd[T](xs: List[T]): (List[T], List[T]) = (xs: @unchecked)
   match {
    case e :: o :: xt =>
      val (es, os) = splitEvenOdd(xt) ((e :: es), (o :: os))
    case Nil => (Nil, Nil)
  }
  def mergeEvenOdd[T](even: List[T], odd: List[T]): List[T] =
   ((even, odd): @unchecked) match {
    case (Nil, Nil) => Nil
    case ((e :: es), (o :: os)) => e :: (o :: mergeEvenOdd(es, os)) }

  def fft(xs: List[Complex]): List[Complex] = xs match {
   case (x :: Nil) => xs case _ => val N = xs.length
   val (even0, odd0) = splitEvenOdd(xs)
   val (even1, odd1) = (fft(even0), fft(odd0))
   val (even2, odd2) = (even1 zip odd1 zipWithIndex) map {
    case ((x, y), k) => val z = omega(k, N) * y (x + z, x - z) } unzip; even2 ::: odd2 }}
```

```
trait FFT { this: Arith with Trig =>
  def omega(k: Int, N: Int): Complex = {
    val kth = -2.0 * k * math.Pi / N Complex(cos(kth), sin(kth)) }
  case class Complex(re: Rep[Double], im: Rep[Double]) {
    def +(that: Complex) = Complex(this.re + that.re, this.im + that.im)
    def -(that: Complex) = Complex(this.re - that.re, this.im - that.im)
    def *(that: Complex) = Complex(this.re * that.re - this.im * that.im,
                                   this.re * that.im + this.im * that.re)
  }
  def splitEvenOdd[T](xs: List[T]): (List[T], List[T]) = (xs: @unchecked)
   match {
    case e :: o :: xt =>
      val (es, os) = splitEvenOdd(xt) ((e :: es), (o :: os))
    case Nil => (Nil, Nil)
  }
  def mergeEvenOdd[T](even: List[T], odd: List[T]): List[T] =
   ((even, odd): @unchecked) match {
    case (Nil, Nil) => Nil
    case ((e :: es), (o :: os)) => e :: (o :: mergeEvenOdd(es, os)) }

  def fft(xs: List[Complex]): List[Complex] = xs match {
   case (x :: Nil) => xs case _ => val N = xs.length
   val (even0, odd0) = splitEvenOdd(xs)
   val (even1, odd1) = (fft(even0), fft(odd0))
   val (even2, odd2) = (even1 zip odd1 zipWithIndex) map {
    case ((x, y), k) => val z = omega(k, N) * y (x + z, x - z) } unzip; even2 ::: odd2 }}
```

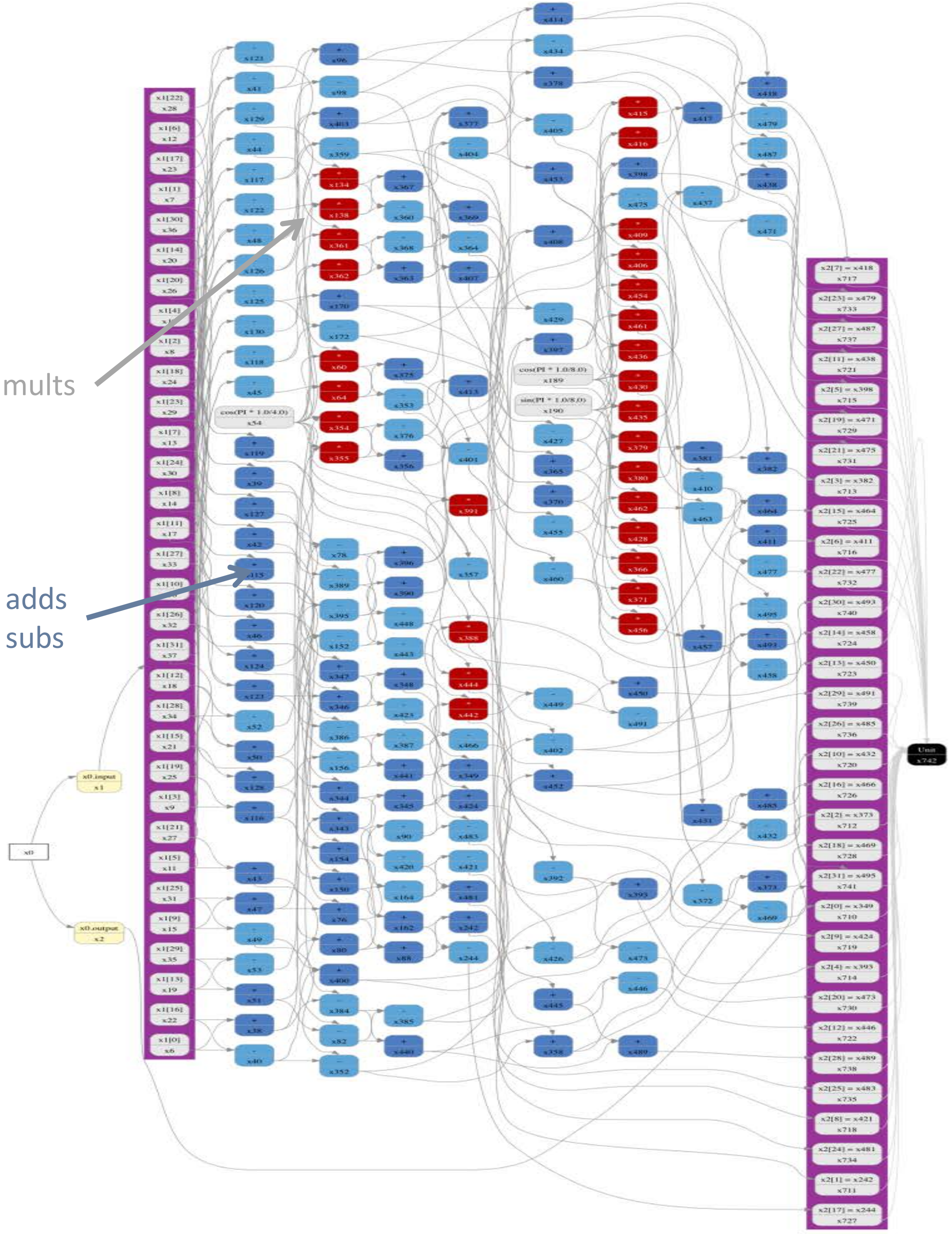
```
trait FFT { this: Arith with Trig =>
  def omega(k: Int, N: Int): Complex = {
    val kth = -2.0 * k * math.Pi / N Complex(cos(kth), sin(kth)) }
  case class Complex(re: Rep[Double], im: Rep[Double]) {
    def +(that: Complex) = Complex(this.re + that.re, this.im + that.im)
    def -(that: Complex) = Complex(this.re - that.re, this.im - that.im)
    def *(that: Complex) = Complex(this.re * that.re - this.im * that.im,
                                 this.re * that.im + this.im * that.re)
  }
  def splitEvenOdd[T](xs: List[T]): (List[T], List[T]) = (xs: @unchecked)
   match {
    case e :: o :: xt =>
      val (es, os) = splitEvenOdd(xt) ((e :: es), (o :: os))
    case Nil => (Nil, Nil)
  }
  def mergeEvenOdd[T](even: List[T], odd: List[T]): List[T] =
   ((even, odd): @unchecked) match {
    case (Nil, Nil) => Nil
    case ((e :: es), (o :: os)) => e :: (o :: mergeEvenOdd(es, os)) }

  def fft(xs: List[Complex]): List[Complex] = xs match {
   case (x :: Nil) => xs case _ => val N = xs.length
   val (even0, odd0) = splitEvenOdd(xs)
   val (even1, odd1) = (fft(even0), fft(odd0))
   val (even2, odd2) = (even1 zip odd1 zipWithIndex) map {
    case ((x, y), k) => val z = omega(k, N) * y (x + z, x - z) } unzip; even2 ::: odd2 }}
```


x0
x0.input
x1
x0.output
x2
x1[22] x28
x1[6] x12
x1[17] x23
x1[1] x7
x1[30] x36
x1[14] x20
x1[20] x26
x1[4] x10
x1[2] x8
x1[18] x24
x1[23] x29
x1[7] x13
x1[24] x30
x1[8] x14
x1[11] x17
x1[27] x33
x1[10] x16
x1[26] x32
x1[31] x37
x1[12] x18
x1[28] x34
x1[15] x21
x1[19] x25
x1[3] x9
x1[21] x27
x1[5] x11
x1[25] x31
x1[9] x15
x1[29] x35
x1[13] x19
x1[16] x22
x1[0] x6
cos(PI * 1.0/4.0) x54
cos(PI * 1.0/8.0) x189
sin(PI * 1.0/8.0) x190
x2[7] = x418 x717
x2[23] = x479 x733
x2[27] = x487 x737
x2[11] = x438 x721
x2[5] = x398 x715
x2[19] = x471 x729
x2[21] = x475 x731
x2[3] = x382 x713
x2[15] = x464 x725
x2[6] = x411 x716
x2[22] = x477 x732
x2[30] = x493 x740
x2[14] = x458 x724
x2[13] = x450 x723
x2[29] = x491 x739
x2[26] = x485 x736
x2[10] = x432 x720
x2[16] = x466 x726
x2[2] = x373 x712
x2[18] = x469 x728
x2[31] = x495 x741
x2[0] = x349 x710
x2[9] = x424 x719
x2[4] = x393 x714
x2[20] = x473 x730
x2[12] = x446 x722
x2[28] = x489 x738
x2[25] = x483 x735
x2[8] = x421 x718
x2[24] = x481 x734
x2[1] = x242 x711
x2[17] = x244 x727
Unit x742



mults

adds
subs



mults

adds
subs

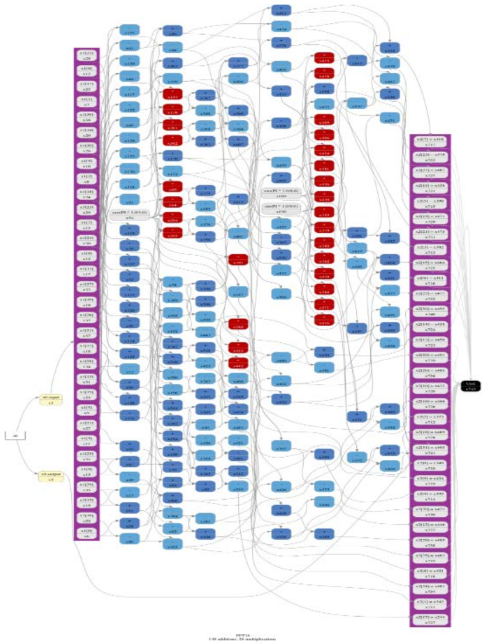
```
typedef struct {
                double* input;
                double* output;
} spiral_t;
const double x708[] = { 1.0, 0.9238795325112867, 0.7071067811865476, 0.38268
const double x709[] = { -0.0, 0.3826834323650898, 0.7071067811865476, 0.9238
void staged(spiral_t* x0) {
double* x2 = x0->output;
double* x1 = x0->input;
double x6 = x1[0];
double x22 = x1[16];
double x38 = x6 + x22;
double x14 = x1[8];
double x30 = x1[24];
double x46 = x14 + x30;
double x343 = x38 + x46;
double x10 = x1[4];
double x26 = x1[20];
double x42 = x10 + x26;
double x18 = x1[12];
double x34 = x1[28];
double x50 = x18 + x34;
double x344 = x42 + x50;
double x345 = x343 + x344;
double x8 = x1[2];
double x24 = x1[18];
double x115 = x8 + x24;
double x16 = x1[10];
double x32 = x1[26];
double x123 = x16 + x32;
double x346 = x115 + x123;
double x12 = x1[6];
double x28 = x1[22];
double x119 = x12 + x28;
double x20 = x1[14];
double x36 = x1[30];
double x127 = x20 + x36;
double x347 = x119 + x127;
double x348 = x346 + x347;
double x349 = x345 + x348;
x2[0] = x349;
double x7 = x1[1];
double x23 = x1[17];
double x39 = x7 + x23;
double x15 = x1[9];
double x31 = x1[25];
double x47 = x15 + x31;
double x76 = x39 + x47;
double x11 = x1[5];
double x27 = x1[21];
double x43 = x11 + x27;
double x19 = x1[13];
double x35 = x1[29];
double x51 = x19 + x35;
```

Selective Staging

```
trait FFT {...
  case class Complex(re: Double, im: Double)
  def fft(…)
...
```

Scala FFT Function

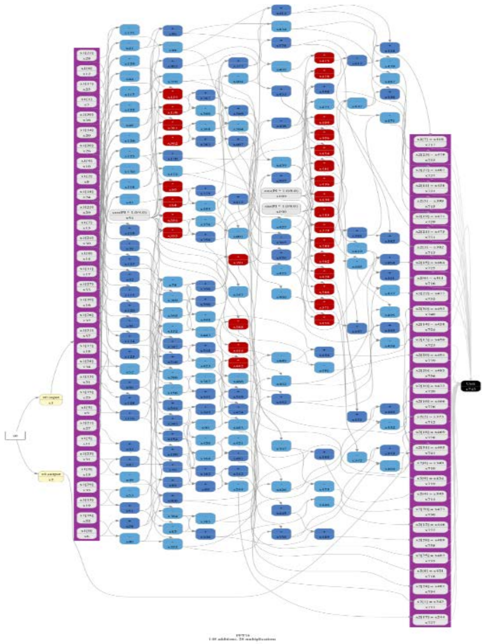
```
trait FFT {...
  case class Complex(re: Rep[Double], im: Rep[Double])
  def fft(…)
...
```



Selective Staging

```
trait FFT {...
  case class Complex(re: Double, im: Double)
  def fft(…)
...
```

Scala FFT Function



```
trait FFT {...
  case class Complex(re: Rep[Double], im: Rep[Double])
  def fft(…)
...
```

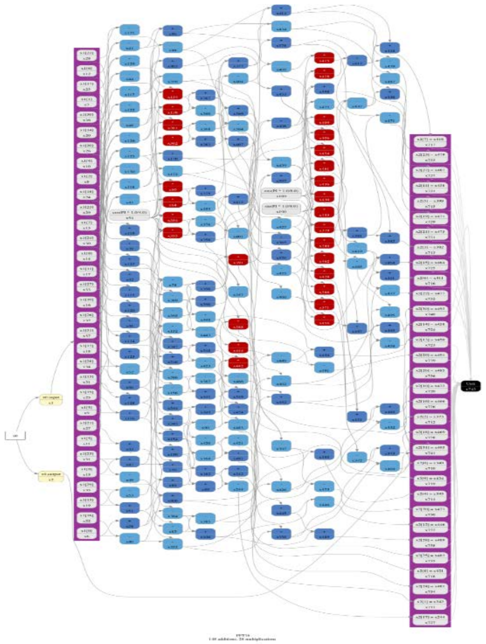
Selective Staging

```
trait FFT {...
  case class Complex(re: Double, im: Double)
  def fft(…)
...
```

```
trait FFT[T] {...
  case class Complex(re: T, im: T)
  def fft(…)
...
```

```
trait FFT {...
  case class Complex(re: Rep[Double], im: Rep[Double])
  def fft(…)
...
```

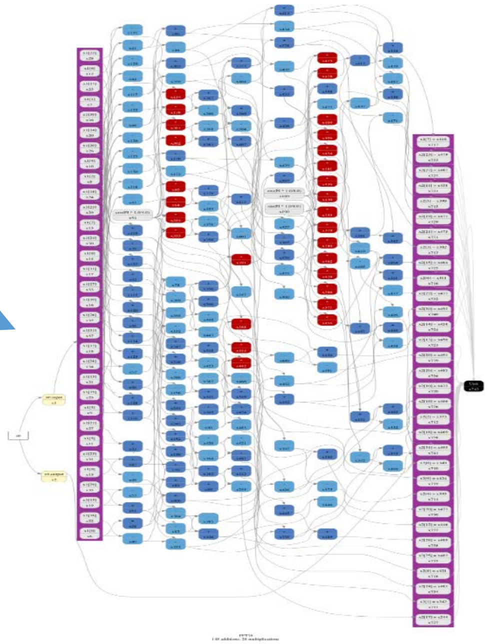
Scala FFT Function



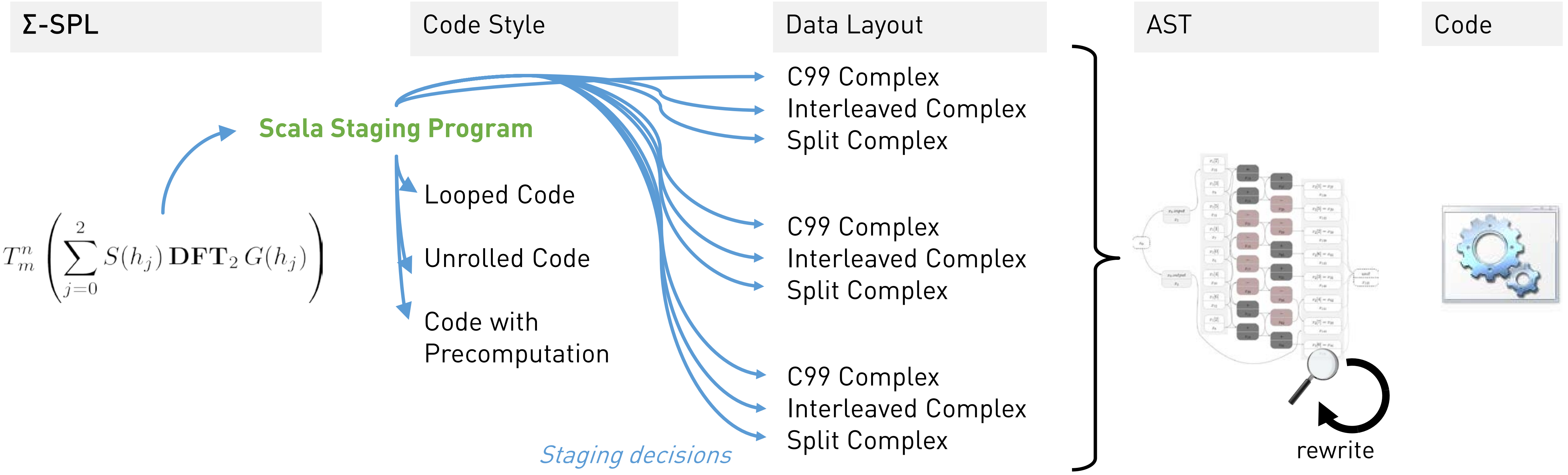
Selective Staging

```
trait FFT {...
  case class Complex(re: Double, im: Double)
  def fft(…)
...

trait FFT[T] {...
  case class Complex(re: T, im: T)
  def fft(…)
...

trait FFT {...
  case class Complex(re: Rep[Double], im: Rep[Double])
  def fft(…)
...
```

Scala FFT Function



Option 3: Staging in Scala



Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n\left(\sum_{j=0}^{2} S(h_j)\,\mathbf{DFT}_2\,G(h_j)\right)$$

Scala Staging Program

Looped Code

Unrolled Code

Code with Precomputation

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

Staging decisions

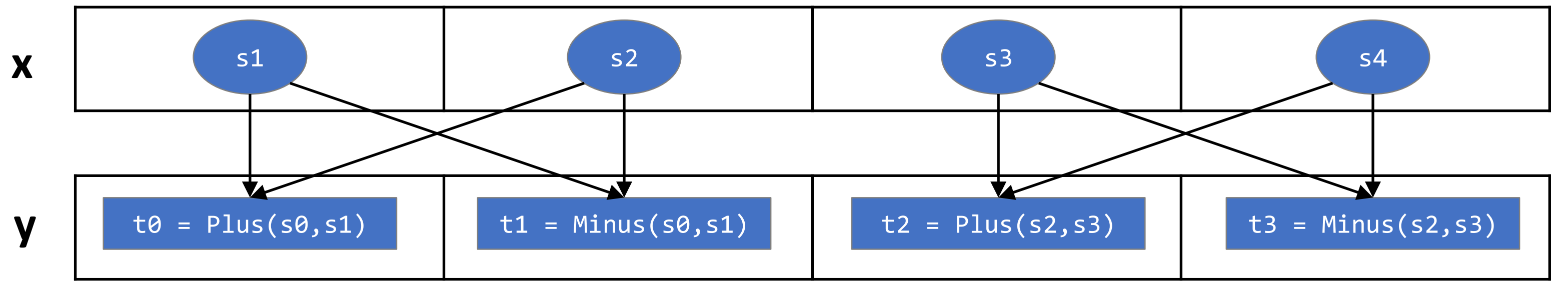
rewrite

Code Style: Unrolled and Scalarized

```
def f(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

Scalars only
Unrolled

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```



x

s1

s2

s3

s4

y

t0 = Plus(s0,s1)

t1 = Minus(s0,s1)

t2 = Plus(s2,s3)

t3 = Minus(s2,s3)

Code Style: Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

Code Style: Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

Code Style: Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

Rep[…].apply(i: Int): T

Code Style: Arrays

i = 0

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

Rep[…].apply(i: Int): T

How to apply() on a Rep?

Code Style: Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

i = 0

Rep[…].apply(i: Int): T

How to apply() on a Rep?

Implementation in **L**ightweight **M**odular **S**taging Library

Code Style: Arrays

i = 0

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

Rep[...].apply(i: Int): T

How to apply() on a Rep?

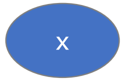
Code Style: Arrays

i = 0

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

Rep[…].apply(i: Int): T

How to apply() on a Rep?



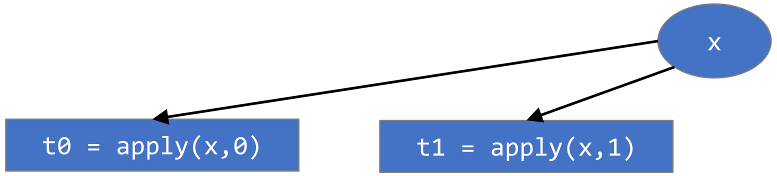
Code Style: Arrays

i = 0

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

Rep[…].apply(i: Int): T

How to apply() on a Rep?



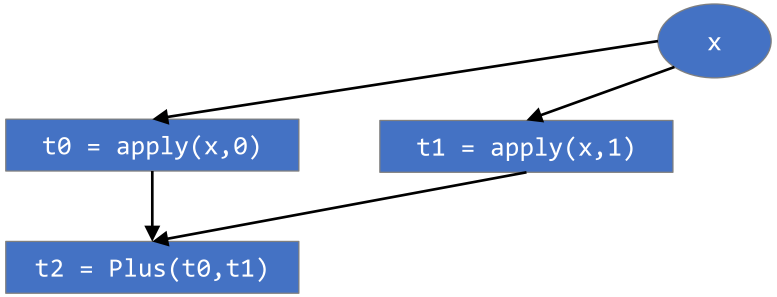
Code Style: Arrays

i = 0

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

Rep[…].apply(i: Int): T

How to apply() on a Rep?



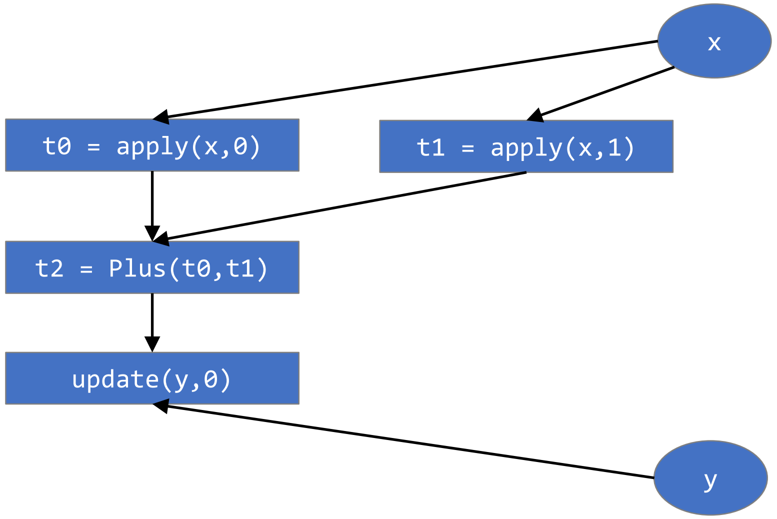
Code Style: Arrays

i = 0

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
```

Rep[…].apply(i: Int): T

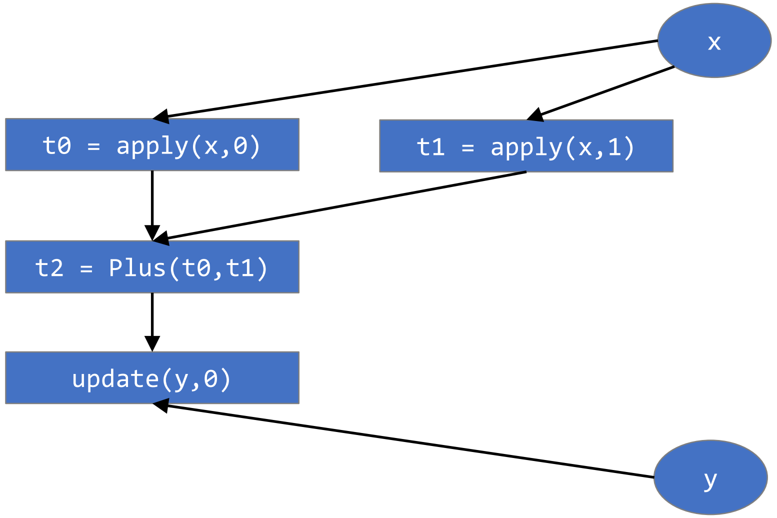
How to apply() on a Rep?



Code Style: Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

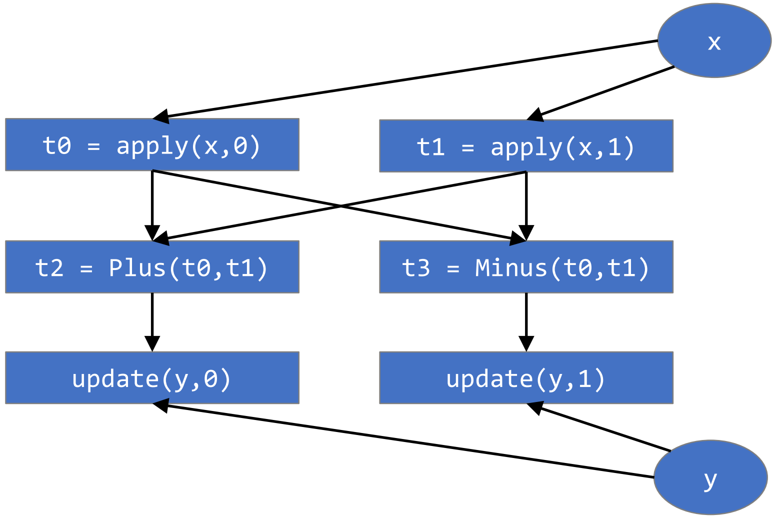
i = 0



Code Style: Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

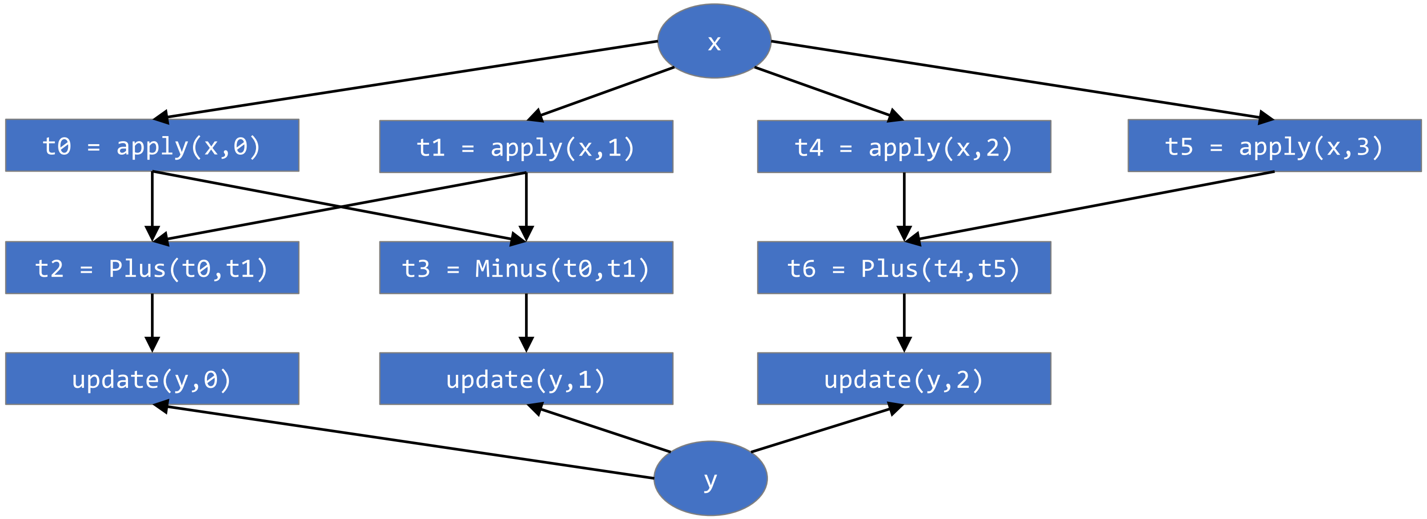
i = 0



Code Style: Arrays

i = 1

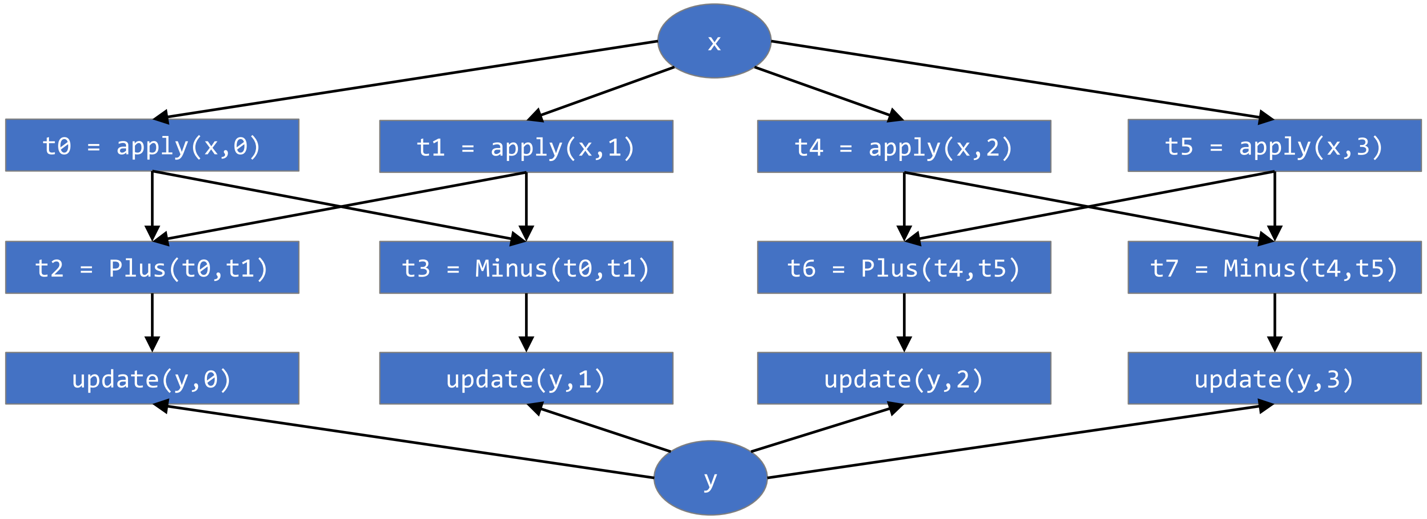
```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```



Code Style: Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

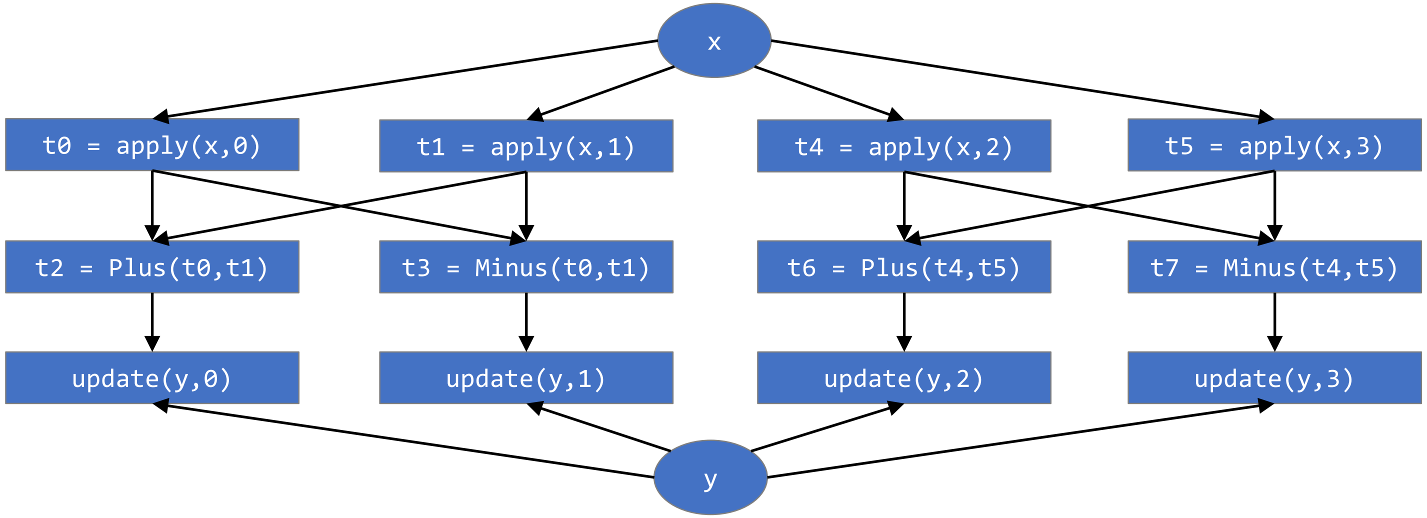
i = 1



Code Style: Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
  for (i <- 0 until 2) {
   y(2*i)   = x(i*2) + x(i*2+1)
   y(2*i+1) = x(i*2) - x(i*2+1)
  }
 }
```

i = 1



x

t0 = apply(x,0)

t1 = apply(x,1)

t4 = apply(x,2)

t5 = apply(x,3)

t2 = Plus(t0,t1)

t3 = Minus(t0,t1)

t6 = Plus(t4,t5)

t7 = Minus(t4,t5)

update(y,0)

update(y,1)

update(y,2)

update(y,3)

y

```
t0 = x[0];
t1 = x[1];
t2 = t0 + t1;
y[0] = t2;
t3 = t0 - t1;
y[1] = t3;
t4 = x[0];
t5 = x[1];
t6 = t4 + x5;
y[0] = t6;
t7 = t4 – x5;
y[3] = t7;
```

Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
```

Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
```

How to apply on a Rep?

Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
    for (i <- 0 until 2: Rep[Range]) {
```

How to apply on a Rep?

Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
```

How to apply on a Rep?

Loop(i, 0, 2)

i

Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
   for (i <- 0 until 2: Rep[Range]) {
    y(2*i)   = x(i*2) + x(i*2+1)
    y(2*i+1) = x(i*2) - x(i*2+1)
   }
  }
```

Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

Loop Body

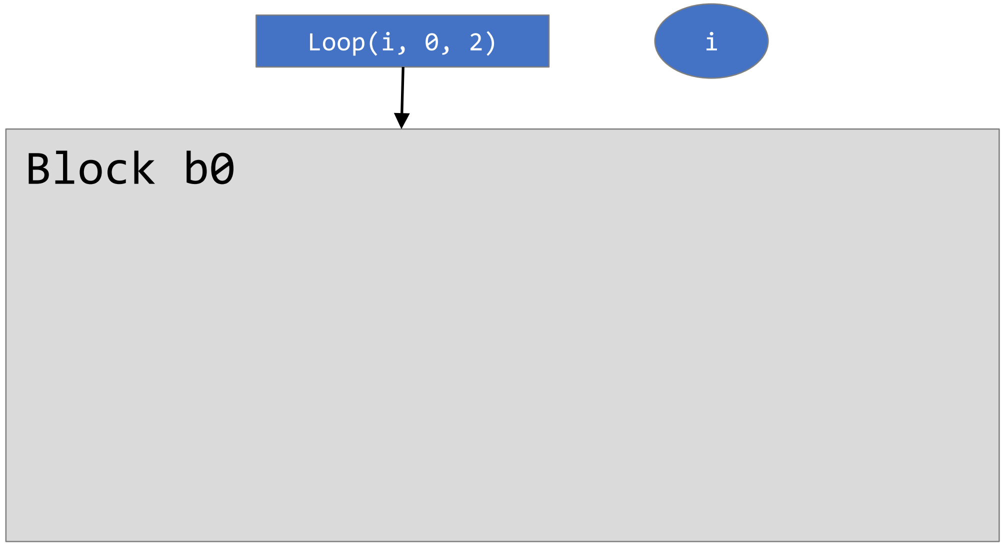
Loop(i, 0, 2)

i

Code Style: Loops & Arrays

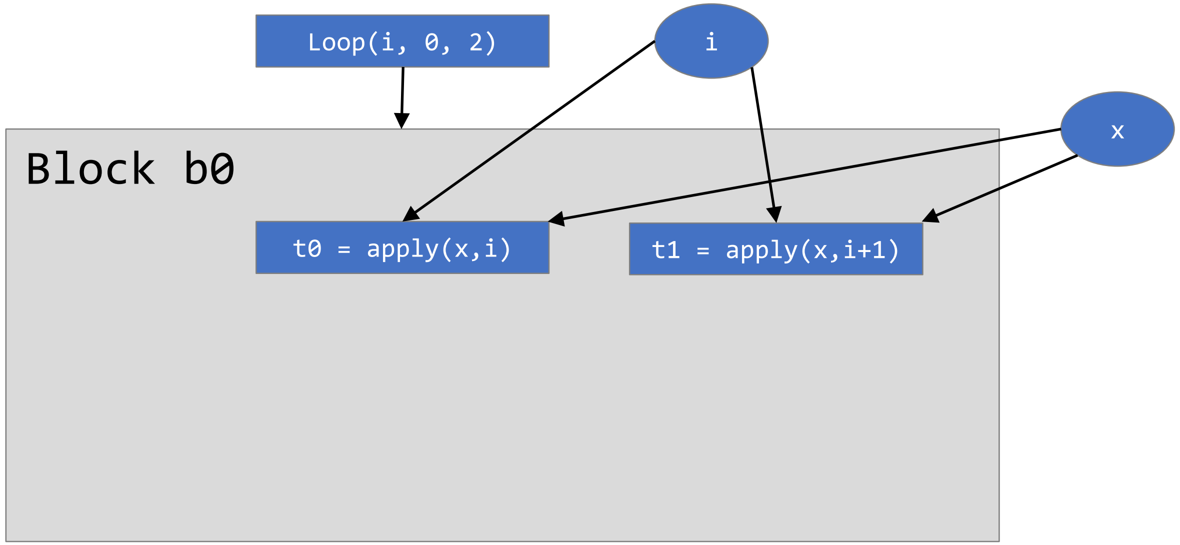
```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
    for (i <- 0 until 2: Rep[Range]) {
      y(2*i)   = x(i*2) + x(i*2+1)
      y(2*i+1) = x(i*2) - x(i*2+1)
    }
  }
```

Loop Body



Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)  } Loop
       y(2*i+1) = x(i*2) - x(i*2+1)  } Body
     }
   }
```



Loop(i, 0, 2)

i

x

Block b0

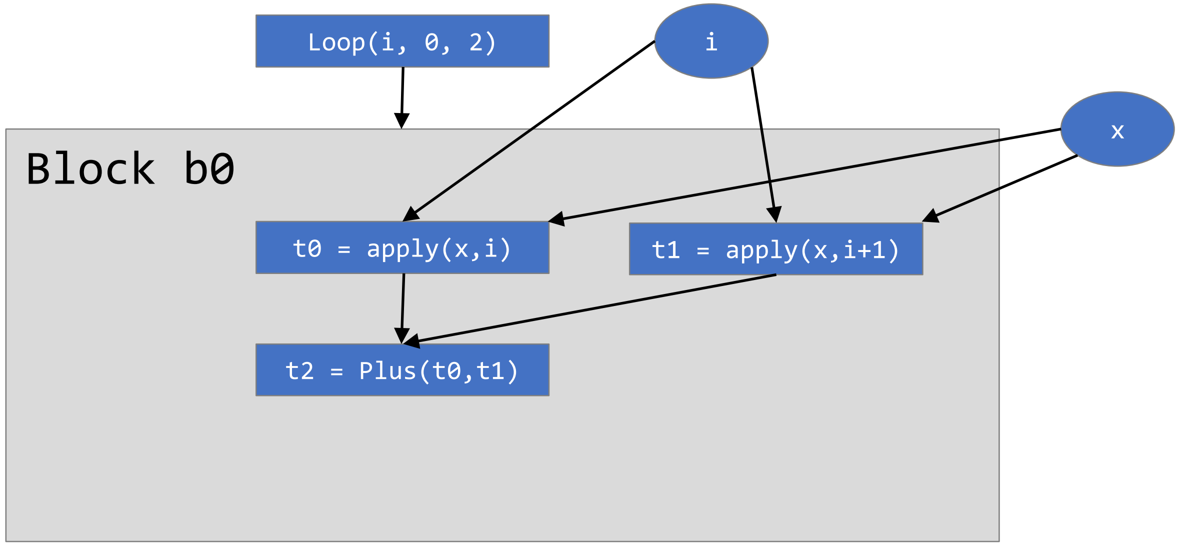
t0 = apply(x,i)

t1 = apply(x,i+1)

Code Style: Loops & Arrays

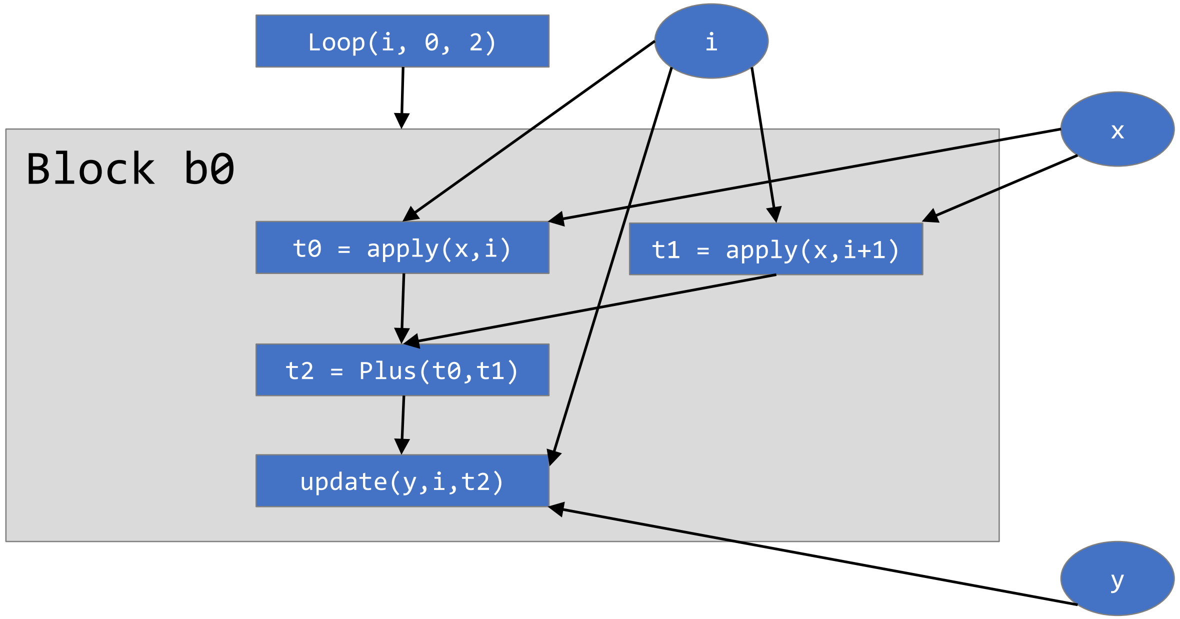
```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

$\Big\}$ Loop Body



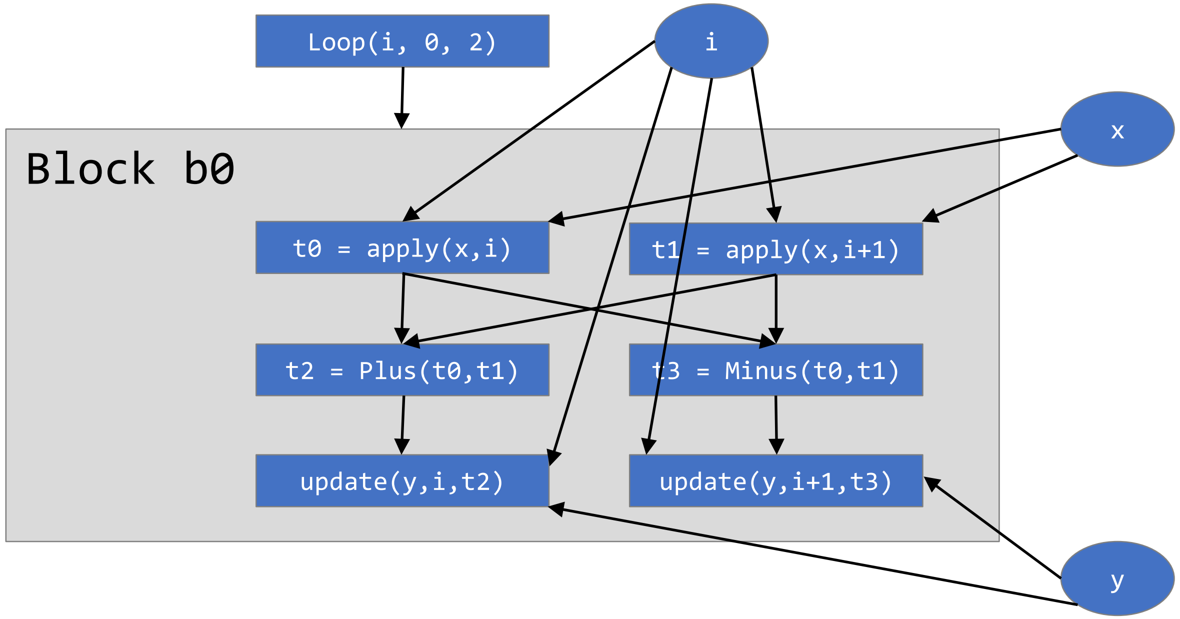
Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)   ⎫ Loop
       y(2*i+1) = x(i*2) - x(i*2+1)   ⎭ Body
     }
   }
```



Code Style: Loops & Arrays

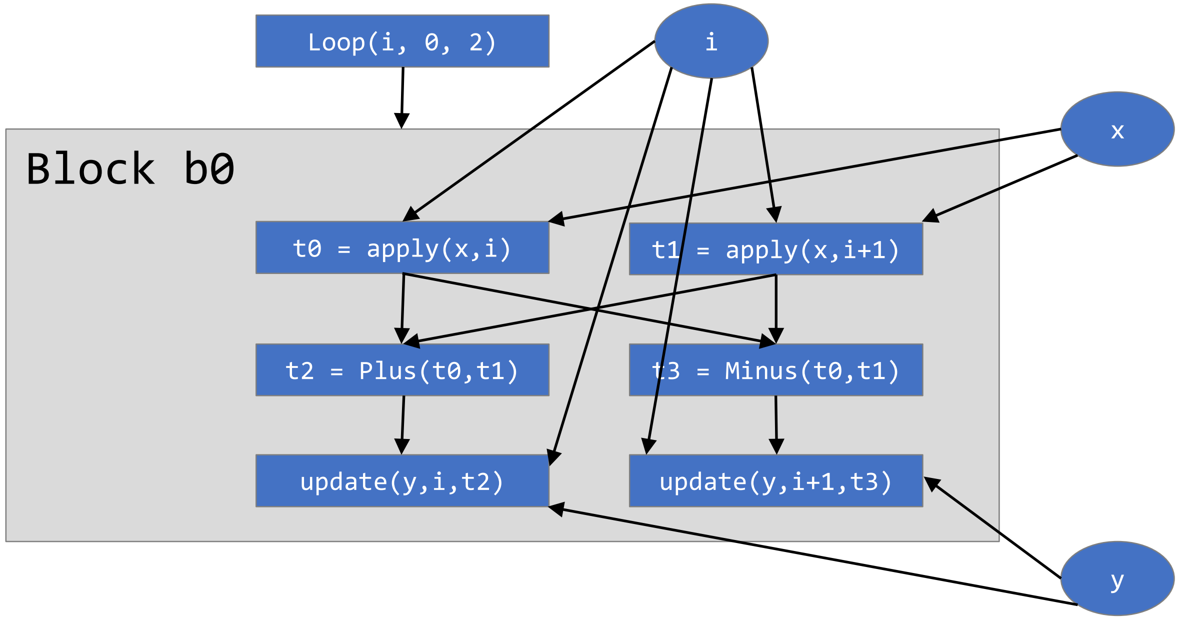
```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)    = x(i*2) + x(i*2+1)   } Loop
       y(2*i+1) = x(i*2) - x(i*2+1)   } Body
     }
   }
```



Code Style: Loops & Arrays

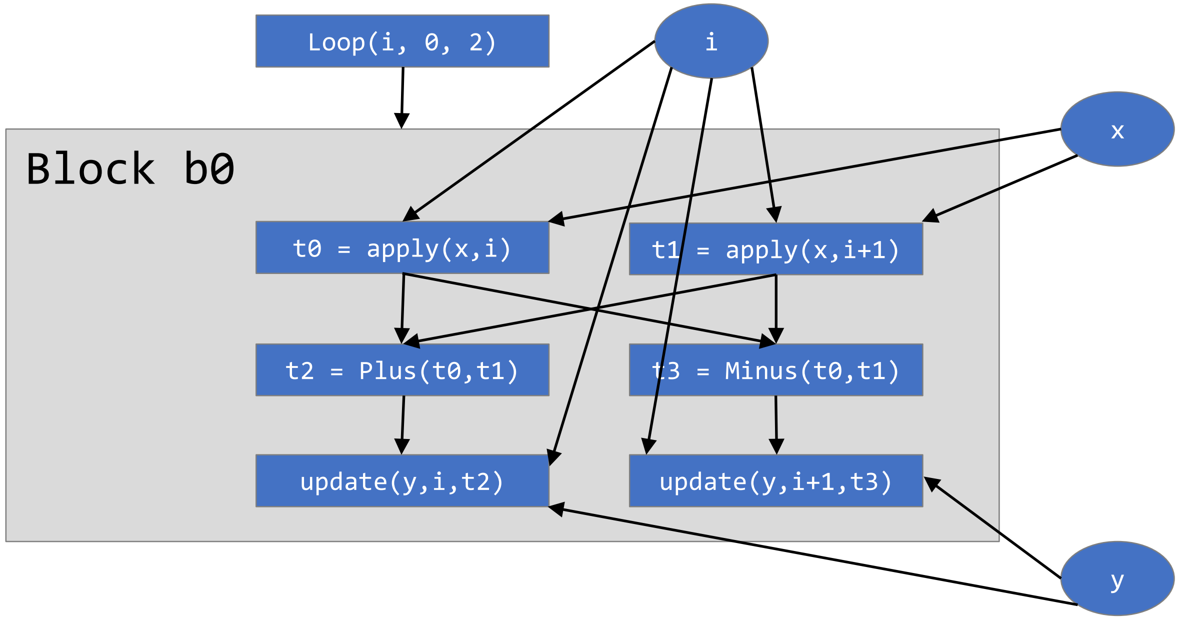
```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

Loop Body



Code Style: Loops & Arrays

```
def f(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)    Loop
       y(2*i+1) = x(i*2) - x(i*2+1)    Body
     }
   }
```

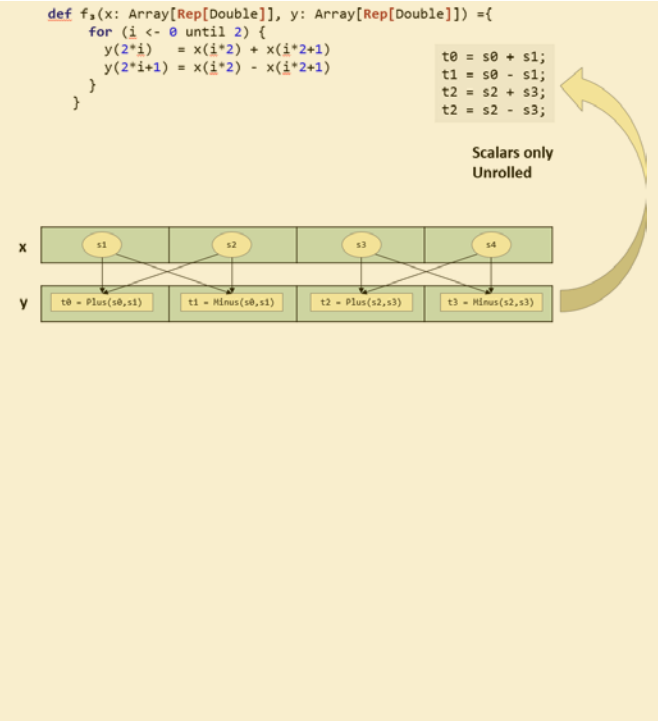


```
for (int i=0;i < 2;i++)
{
    t0 = x[i];
    t1 = x[i+1];
    t2 = t0 + t1;
    y[i] = t2;
    t3 = t0 - t1;
    y[i+1] = t3;
}
```

Abstraction over Code Types

Abstraction over Code Types

Unrolled, scalarized



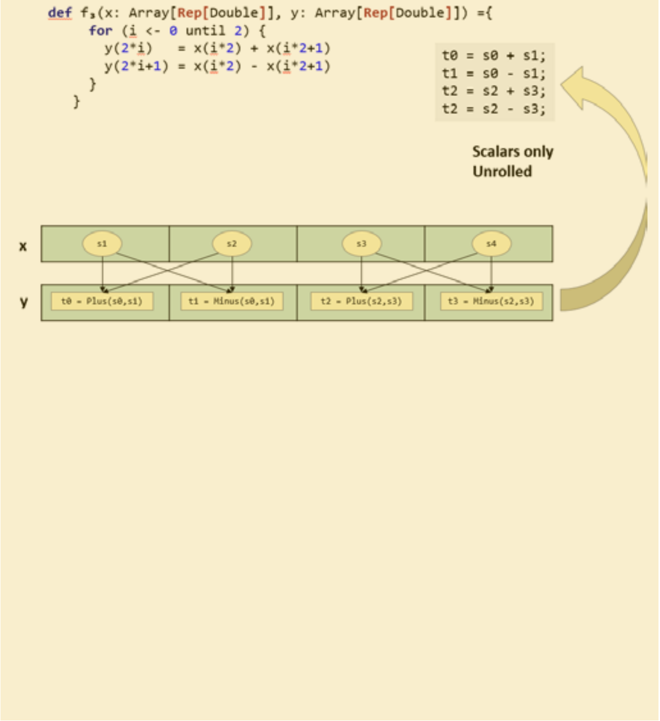
```
def f3(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
    for (i <- 0 until 2) {
      y(2*i)   = x(i*2) + x(i*2+1)
      y(2*i+1) = x(i*2) - x(i*2+1)
    }
  }
```

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

Scalars only
Unrolled

Abstraction over Code Types

Unrolled, scalarized

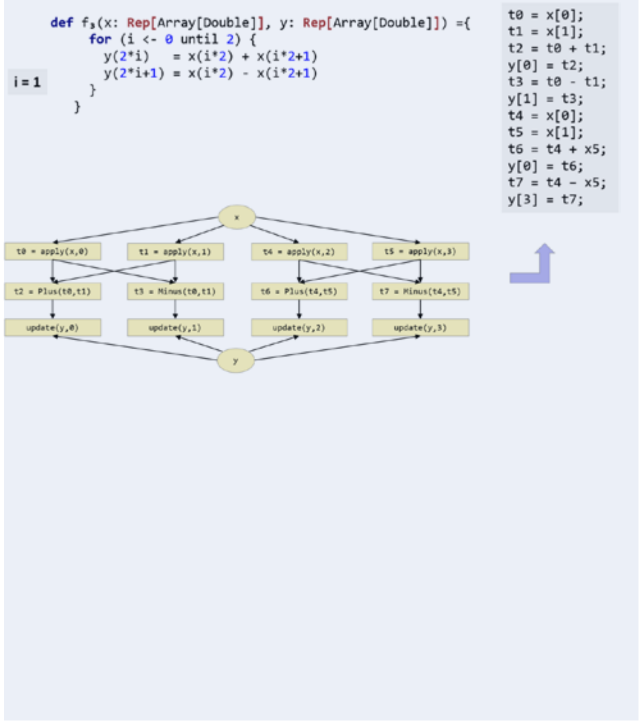


```
def f₃(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

Scalars only
Unrolled

Unrolled, Arrays



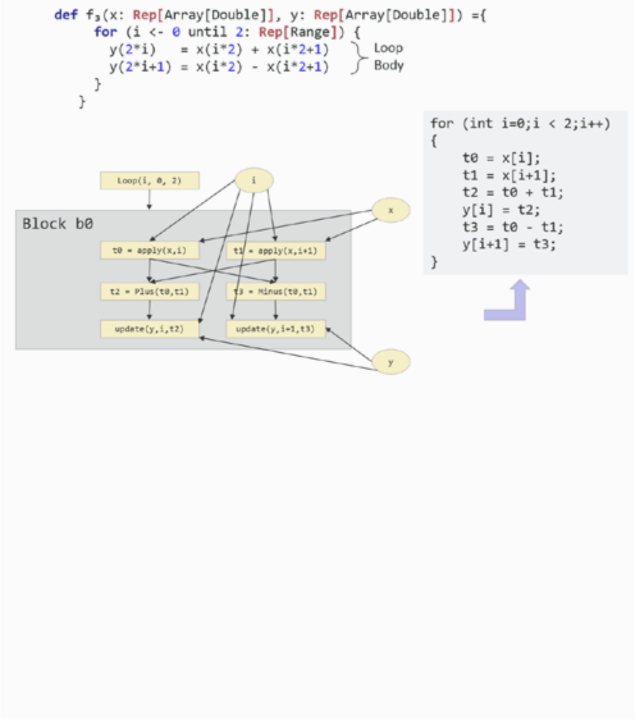
```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

i = 1

```
t0 = x[0];
t1 = x[1];
t2 = t0 + t1;
y[0] = t2;
t3 = t0 - t1;
y[1] = t3;
t4 = x[0];
t5 = x[1];
t6 = t4 + x5;
y[0] = t6;
t7 = t4 – x5;
y[3] = t7;
```

Abstraction over Code Types

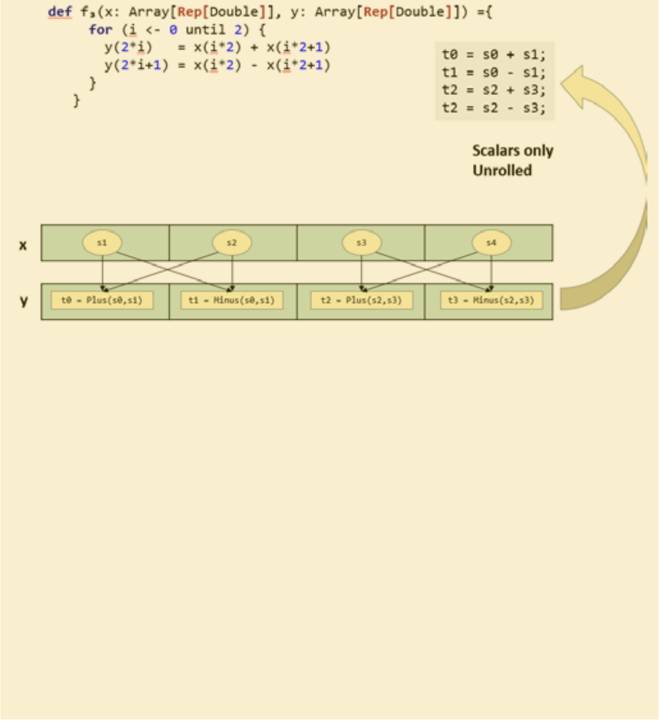
Looped, Arrays



```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
    for (i <- 0 until 2: Rep[Range]) {
      y(2*i)   = x(i*2) + x(i*2+1)
      y(2*i+1) = x(i*2) - x(i*2+1)
    }
  }
```

```
for (int i=0;i < 2;i++)
{
    t0 = x[i];
    t1 = x[i+1];
    t2 = t0 + t1;
    y[i] = t2;
    t3 = t0 - t1;
    y[i+1] = t3;
}
```

Unrolled, scalarized

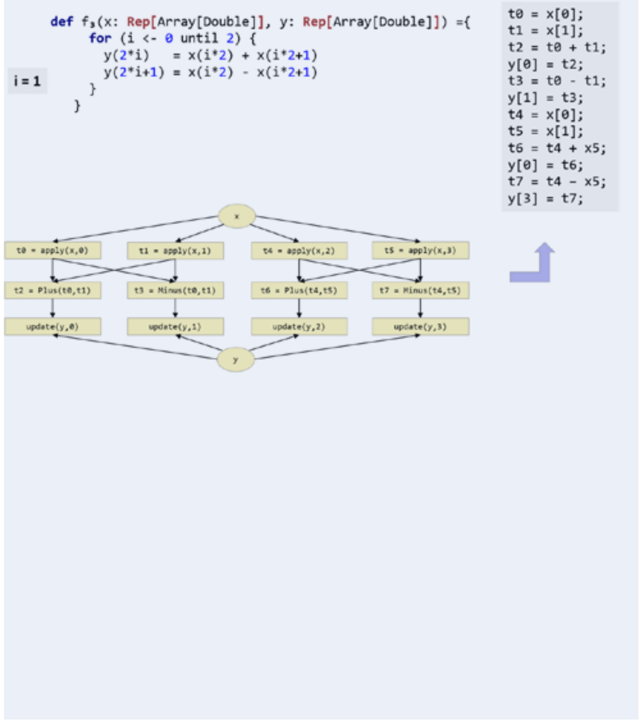


```
def f₃(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
    for (i <- 0 until 2) {
      y(2*i)   = x(i*2) + x(i*2+1)
      y(2*i+1) = x(i*2) - x(i*2+1)
    }
  }
```

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

Scalars only
Unrolled

Unrolled, Arrays



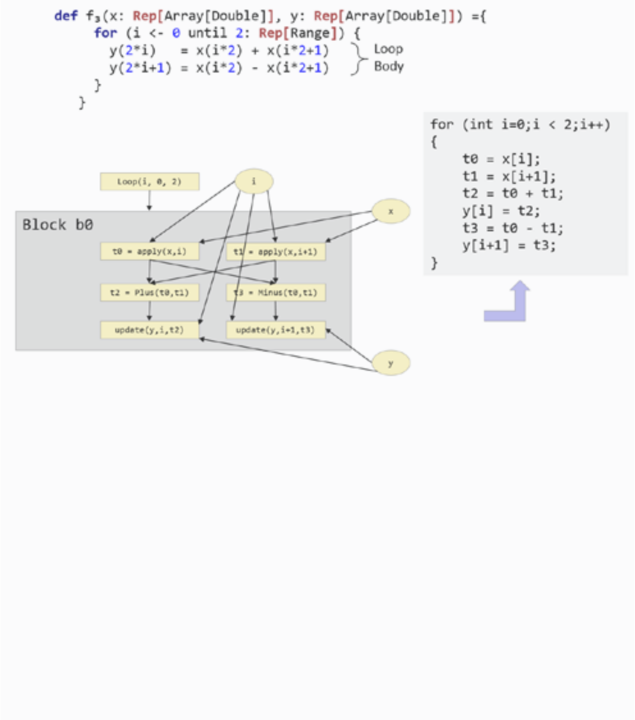
```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
    for (i <- 0 until 2) {
      y(2*i)   = x(i*2) + x(i*2+1)
      y(2*i+1) = x(i*2) - x(i*2+1)
    }
  }
```

i = 1

```
t0 = x[0];
t1 = x[1];
t2 = t0 + t1;
y[0] = t2;
t3 = t0 - t1;
y[1] = t3;
t4 = x[0];
t5 = x[1];
t6 = t4 + x5;
y[0] = t6;
t7 = t4 – x5;
y[3] = t7;
```

Abstraction over Code Types

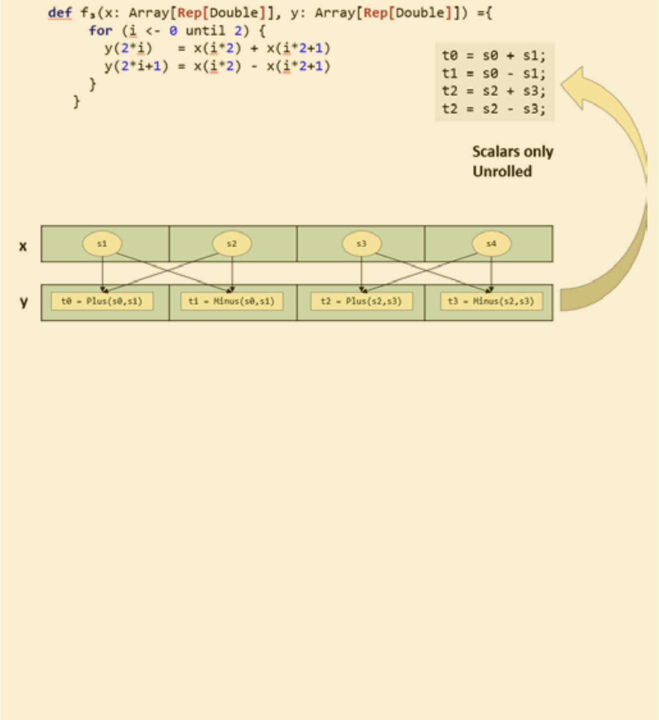
Looped, Arrays



```
def f3(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
for (int i=0;i < 2;i++)
{
    t0 = x[i];
    t1 = x[i+1];
    t2 = t0 + t1;
    y[i] = t2;
    t3 = t0 - t1;
    y[i+1] = t3;
}
```

Unrolled, scalarized

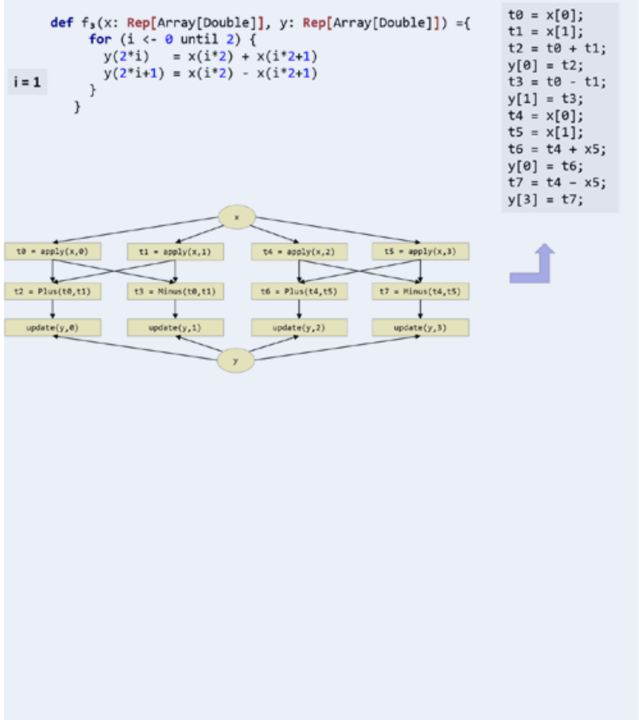


```
def f3(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

Scalars only
Unrolled

Unrolled, Arrays



```
def f3(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

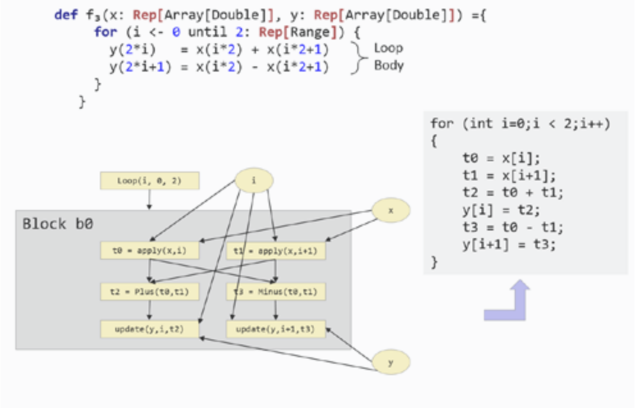
i = 1

```
t0 = x[0];
t1 = x[1];
t2 = t0 + t1;
y[0] = t2;
t3 = t0 - t1;
y[1] = t3;
t4 = x[0];
t5 = x[1];
t6 = t4 + x5;
y[0] = t6;
t7 = t4 – x5;
y[3] = t7;
```

```
def f[L[_],A[_],T](looptype: L, x: A[Array[T]], y: A[Array[T]]) ={
     for (i <- 0 until 2: L[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1)= x(i*2) - x(i*2+1)
}}
```

Abstraction over Code Types

Looped, Arrays

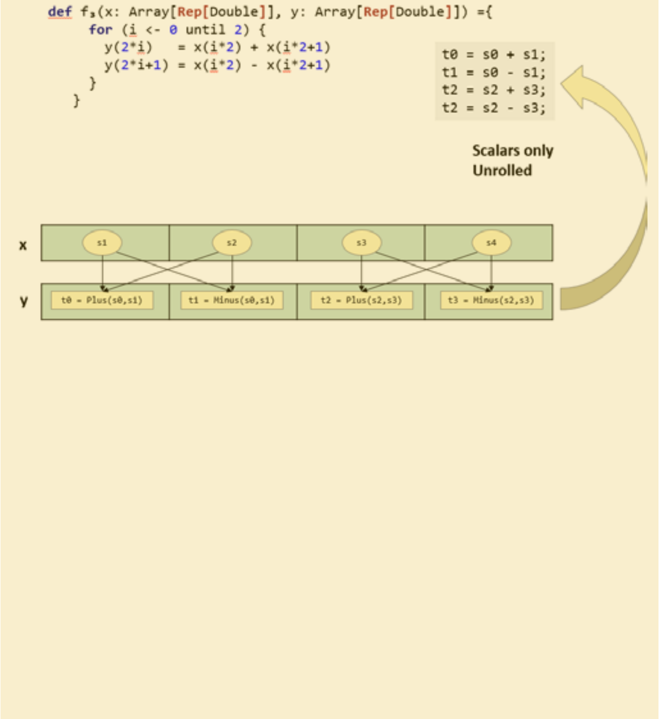


```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)   } Loop
       y(2*i+1) = x(i*2) - x(i*2+1)   } Body
     }
   }
```

```
for (int i=0;i < 2;i++)
{
    t0 = x[i];
    t1 = x[i+1];
    t2 = t0 + t1;
    y[i] = t2;
    t3 = t0 - t1;
    y[i+1] = t3;
}
```

```
f₃(
Rep,
Rep[Array[Double]],
Rep[Array[Double]]
)
```

Unrolled, scalarized

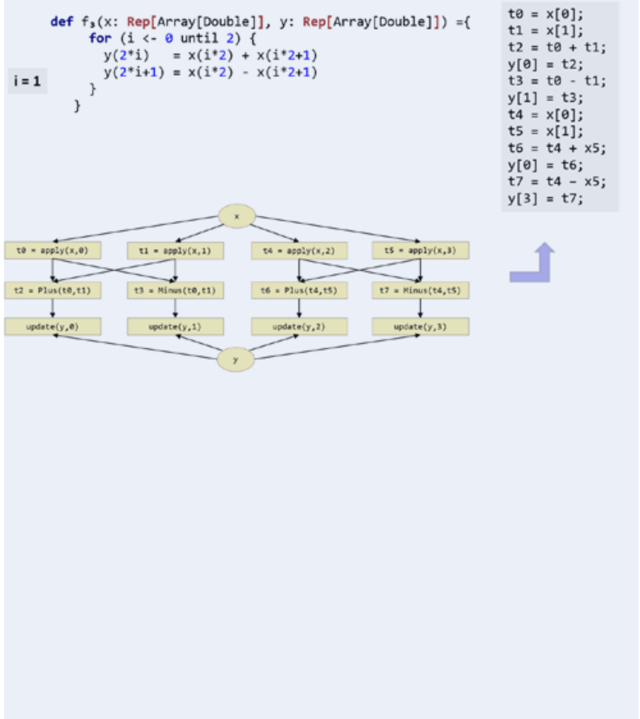


```
def f₃(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

Scalars only
Unrolled

Unrolled, Arrays



```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

i = 1

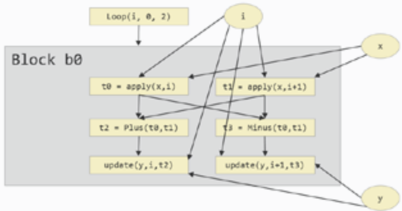
```
t0 = x[0];
t1 = x[1];
t2 = t0 + t1;
y[0] = t2;
t3 = t0 - t1;
y[1] = t3;
t4 = x[0];
t5 = x[1];
t6 = t4 + x5;
y[0] = t6;
t7 = t4 – x5;
y[3] = t7;
```

```
def f[L[_],A[_],T](looptype: L, x: A[Array[T]], y: A[Array[T]]) ={
     for (i <- 0 until 2: L[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1)= x(i*2) - x(i*2+1)
}}
```

Abstraction over Code Types

Looped, Arrays

```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)   ⎫ Loop
       y(2*i+1) = x(i*2) - x(i*2+1)   ⎭ Body
     }
   }
```

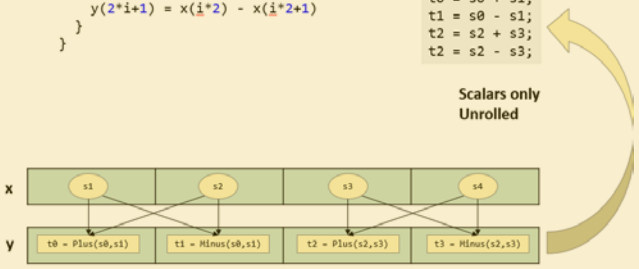


```
for (int i=0;i < 2;i++)
{
    t0 = x[i];
    t1 = x[i+1];
    t2 = t0 + t1;
    y[i] = t2;
    t3 = t0 - t1;
    y[i+1] = t3;
}
```

```
f₃(
Rep,
Rep[Array[Double]],
Rep[Array[Double]]
)
```

Unrolled, scalarized

```
def f₃(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```



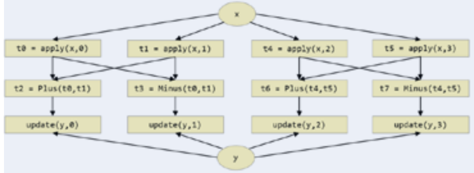
```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

Scalars only
Unrolled

Unrolled, Arrays

```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

i = 1



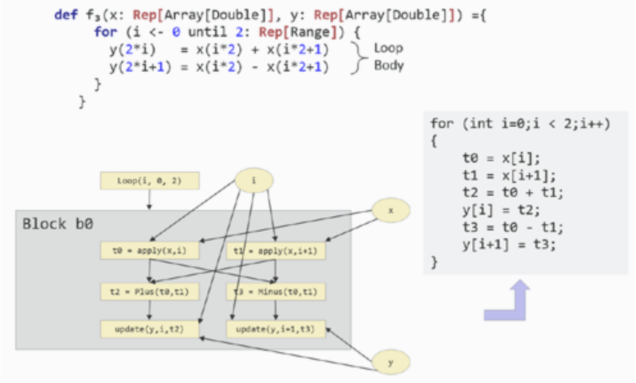
```
t0 = x[0];
t1 = x[1];
t2 = t0 + t1;
y[0] = t2;
t3 = t0 - t1;
y[1] = t3;
t4 = x[0];
t5 = x[1];
t6 = t4 + x5;
y[0] = t6;
t7 = t4 – x5;
y[3] = t7;
```

```
def f[L[_],A[_],T](looptype: L, x: A[Array[T]], y: A[Array[T]]) ={
     for (i <- 0 until 2: L[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1)= x(i*2) - x(i*2+1)
}}
```

```
type NoRep[T] = T
```

Abstraction over Code Types

Looped, Arrays

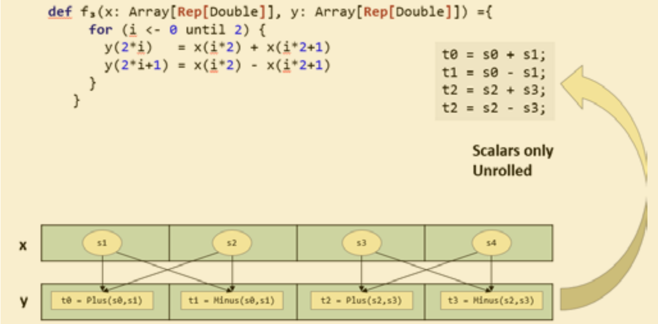


```
def f3(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
    for (i <- 0 until 2: Rep[Range]) {
      y(2*i)   = x(i*2) + x(i*2+1)    } Loop
      y(2*i+1) = x(i*2) - x(i*2+1)    } Body
    }
  }
```

```
for (int i=0;i < 2;i++)
{
    t0 = x[i];
    t1 = x[i+1];
    t2 = t0 + t1;
    y[i] = t2;
    t3 = t0 - t1;
    y[i+1] = t3;
}
```

```
f3(
Rep,
Rep[Array[Double]],
Rep[Array[Double]]
)
```

Unrolled, scalarized

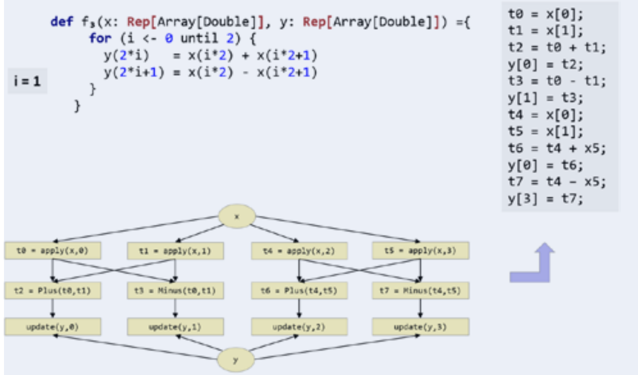


```
def f3(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
    for (i <- 0 until 2) {
      y(2*i)   = x(i*2) + x(i*2+1)
      y(2*i+1) = x(i*2) - x(i*2+1)
    }
  }
```

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

Scalars only
Unrolled

Unrolled, Arrays



```
def f3(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
    for (i <- 0 until 2) {
      y(2*i)   = x(i*2) + x(i*2+1)
      y(2*i+1) = x(i*2) - x(i*2+1)
    }
  }
```

i = 1

```
t0 = x[0];
t1 = x[1];
t2 = t0 + t1;
y[0] = t2;
t3 = t0 - t1;
y[1] = t3;
t4 = x[0];
t5 = x[1];
t6 = t4 + x5;
y[0] = t6;
t7 = t4 – x5;
y[3] = t7;
```

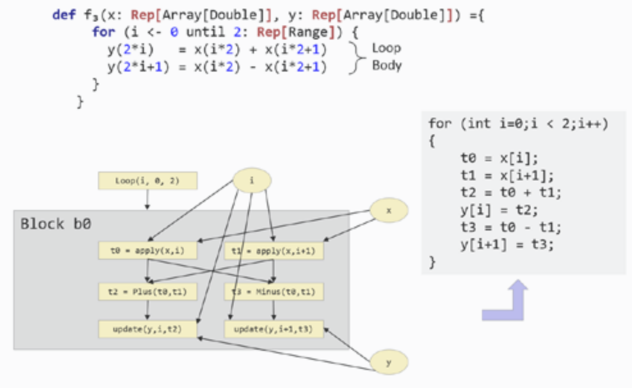
```
f3(
NoRep,
Rep[Array[Double]],
Rep[Array[Double]]
)
```

```
def f[L[_],A[_],T](looptype: L, x: A[Array[T]], y: A[Array[T]]) ={
     for (i <- 0 until 2: L[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1)= x(i*2) - x(i*2+1)
}}
```

```
type NoRep[T] = T
```

Abstraction over Code Types

Looped, Arrays

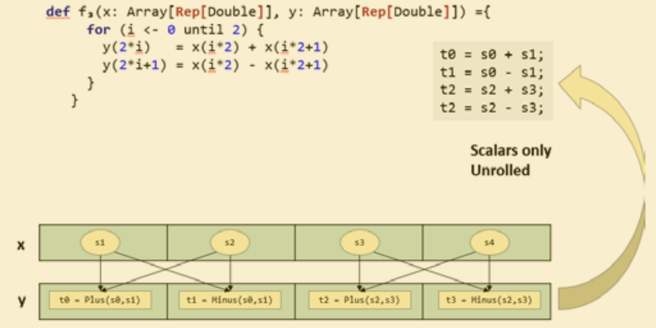


```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
for (int i=0;i < 2;i++)
{
    t0 = x[i];
    t1 = x[i+1];
    t2 = t0 + t1;
    y[i] = t2;
    t3 = t0 - t1;
    y[i+1] = t3;
}
```

```
f₃(
Rep,
Rep[Array[Double]],
Rep[Array[Double]]
)
```

Unrolled, scalarized

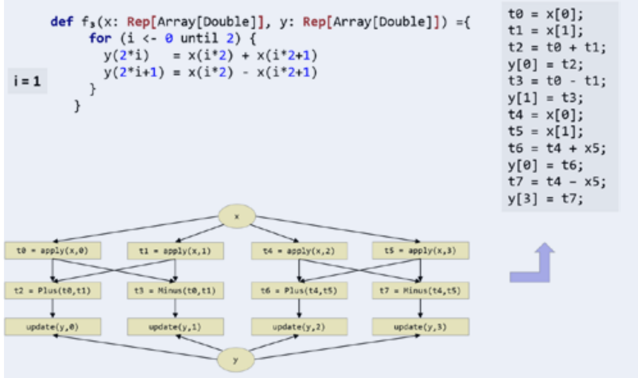


```
def f₃(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

```
f₃(
NoRep,
NoRep[Array[Rep[Double]]],
NoRep[Array[Rep[Double]]]
)
```

Unrolled, Arrays



```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
t0 = x[0];
t1 = x[1];
t2 = t0 + t1;
y[0] = t2;
t3 = t0 - t1;
y[1] = t3;
t4 = x[0];
t5 = x[1];
t6 = t4 + x5;
y[0] = t6;
t7 = t4 – x5;
y[3] = t7;
```

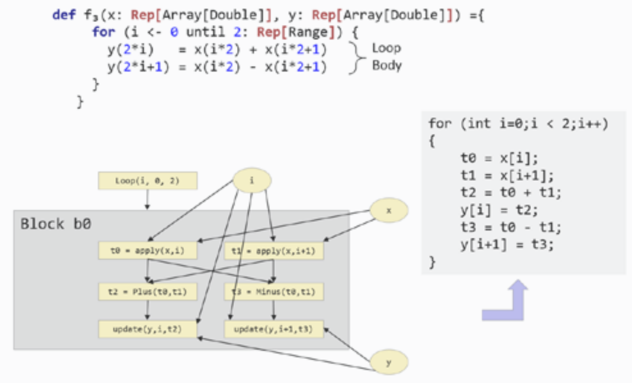
```
f₃(
NoRep,
Rep[Array[Double]],
Rep[Array[Double]]
)
```

```
def f[L[_],A[_],T](looptype: L, x: A[Array[T]], y: A[Array[T]]) ={
     for (i <- 0 until 2: L[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1)= x(i*2) - x(i*2+1)
}}
```

```
type NoRep[T] = T
```

Abstraction over Code Types

Looped, Arrays

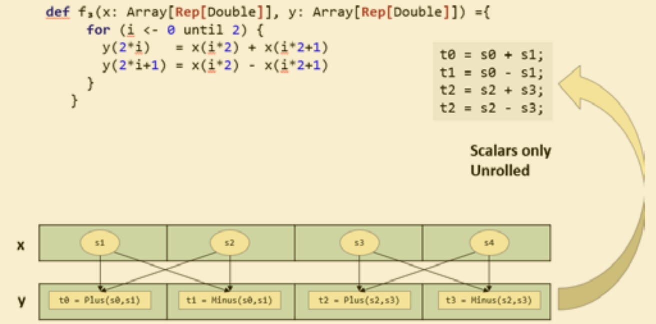


```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2: Rep[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
for (int i=0;i < 2;i++)
{
    t0 = x[i];
    t1 = x[i+1];
    t2 = t0 + t1;
    y[i] = t2;
    t3 = t0 - t1;
    y[i+1] = t3;
}
```

```
f₃(
Rep,
Rep[Array[Double]],
Rep[Array[Double]]
)
```

Unrolled, scalarized

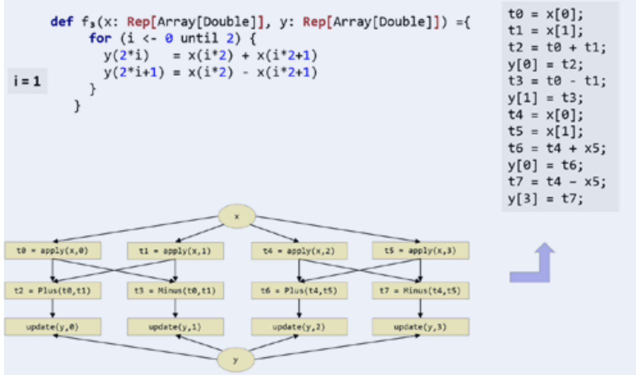


```
def f₃(x: Array[Rep[Double]], y: Array[Rep[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
t0 = s0 + s1;
t1 = s0 - s1;
t2 = s2 + s3;
t2 = s2 - s3;
```

```
f₃(
NoRep,
NoRep[Array[Rep[Double]]],
NoRep[Array[Rep[Double]]]
)
```

Unrolled, Arrays



```
def f₃(x: Rep[Array[Double]], y: Rep[Array[Double]]) ={
     for (i <- 0 until 2) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1) = x(i*2) - x(i*2+1)
     }
   }
```

```
t0 = x[0];
t1 = x[1];
t2 = t0 + t1;
y[0] = t2;
t3 = t0 - t1;
y[1] = t3;
t4 = x[0];
t5 = x[1];
t6 = t4 + x5;
y[0] = t6;
t7 = t4 – x5;
y[3] = t7;
```

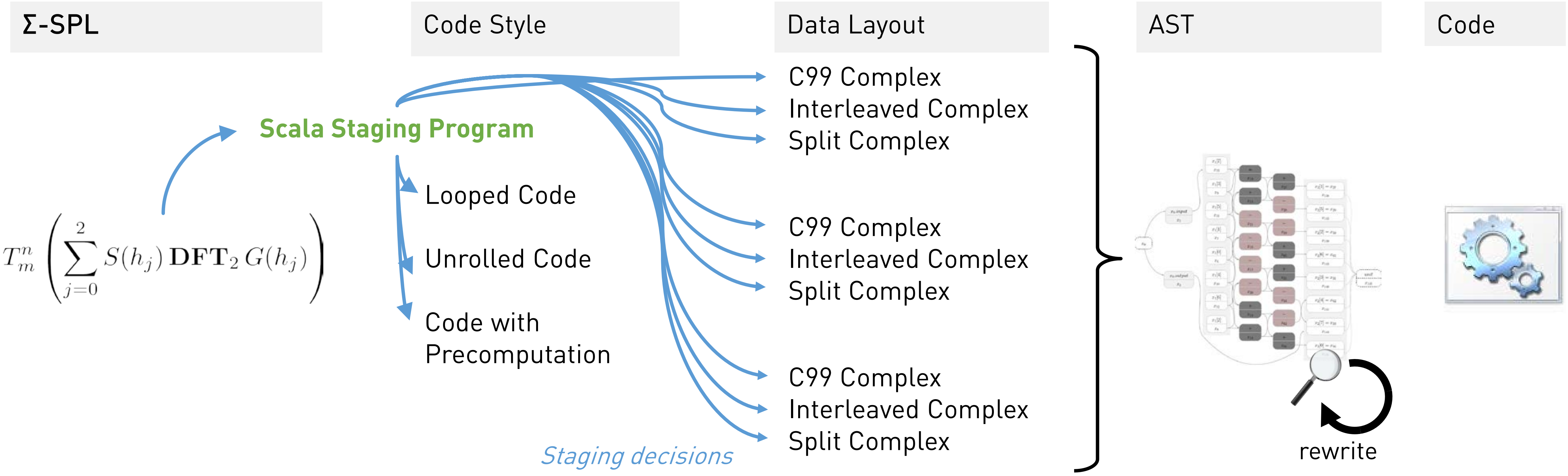
```
f₃(
NoRep,
Rep[Array[Double]],
Rep[Array[Double]]
)
```

```
def f[L[_],A[_],T](looptype: L, x: A[Array[T]], y: A[Array[T]]) ={
     for (i <- 0 until 2: L[Range]) {
       y(2*i)   = x(i*2) + x(i*2+1)
       y(2*i+1)= x(i*2) - x(i*2+1)
}}
```

```
type NoRep[T] = T
```

*Typeclasses omitted for simplicity

Option 3: Staging in Scala



Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right)$$

Scala Staging Program

Looped Code

Unrolled Code

Code with Precomputation

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

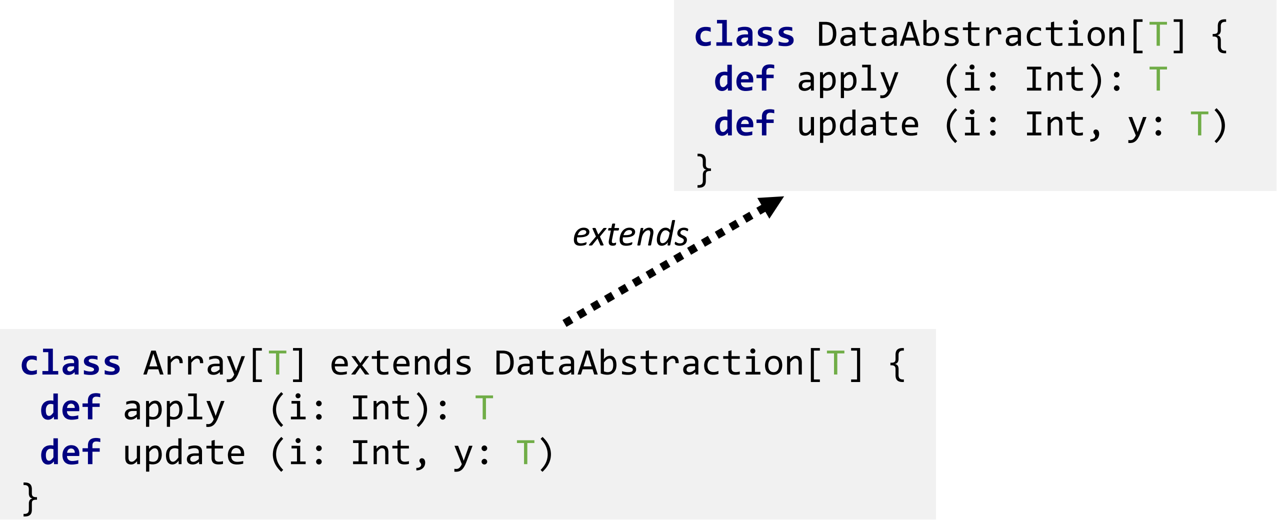
Staging decisions

rewrite

Abstraction over Data Layout

```
class DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

Abstraction over Data Layout

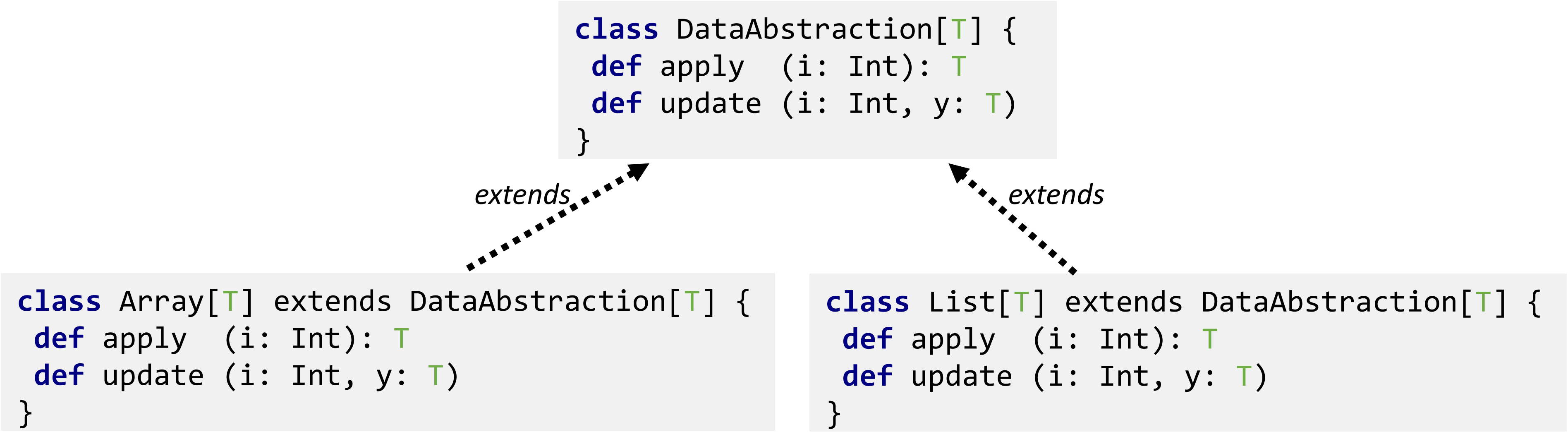


```
class DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

extends

```
class Array[T] extends DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

Abstraction over Data Layout

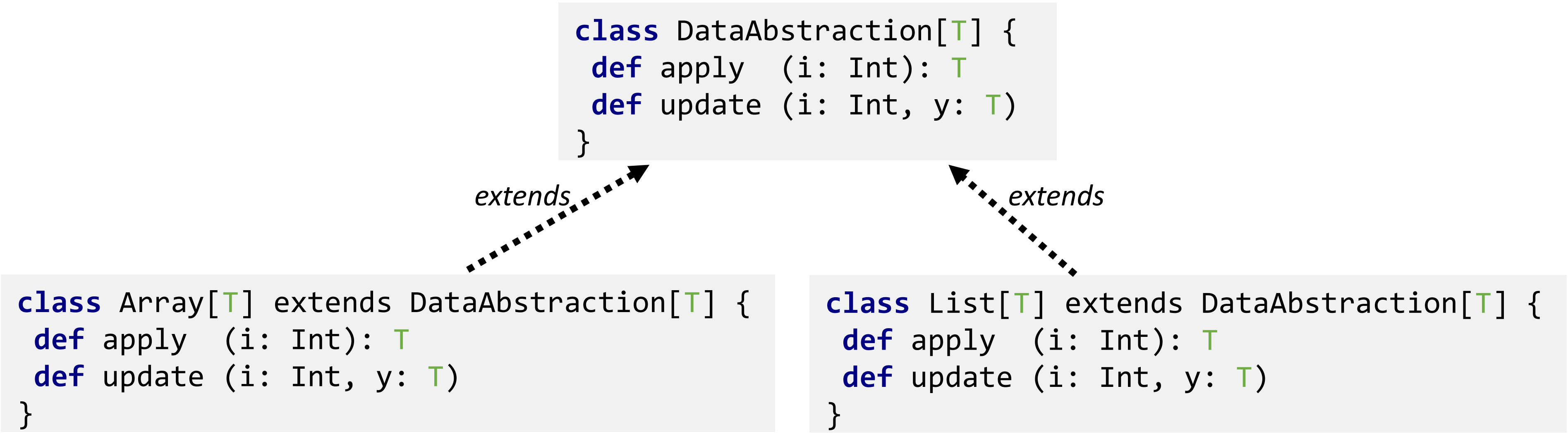


```
class DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

```
class Array[T] extends DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

```
class List[T] extends DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

Abstraction over Data Layout



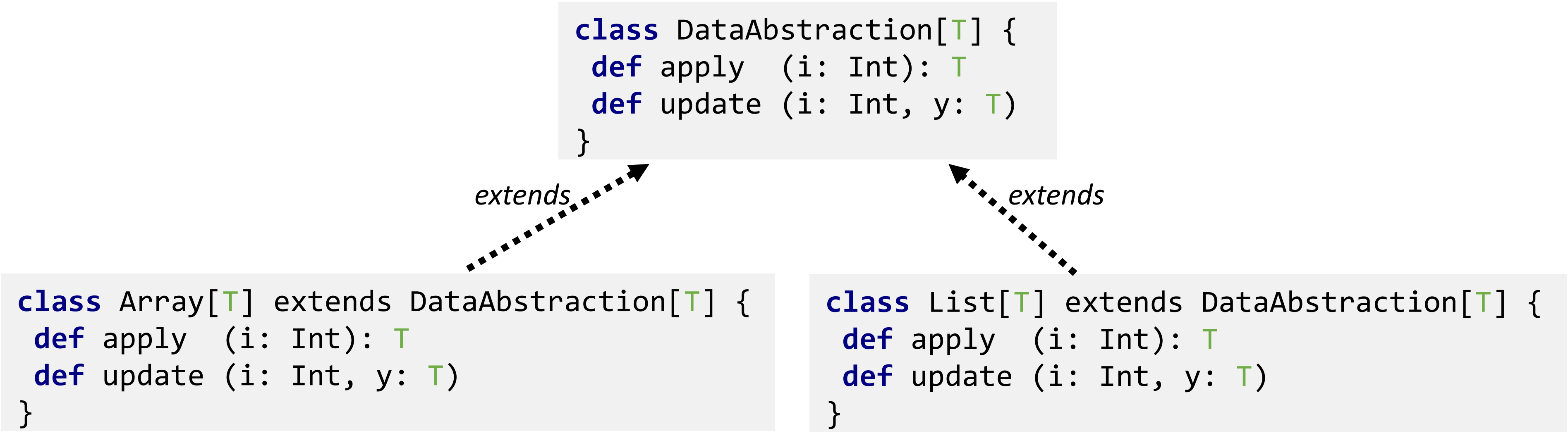
```
class DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

```
class Array[T] extends DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

```
class List[T] extends DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

```
def f[T](x: DataAbstraction, y: DataAbstraction) {
 x(0) = y(0)
}
```

Abstraction over Data Layout



```
class DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

extends

extends

```
class Array[T] extends DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

```
class List[T] extends DataAbstraction[T] {
 def apply  (i: Int): T
 def update (i: Int, y: T)
}
```

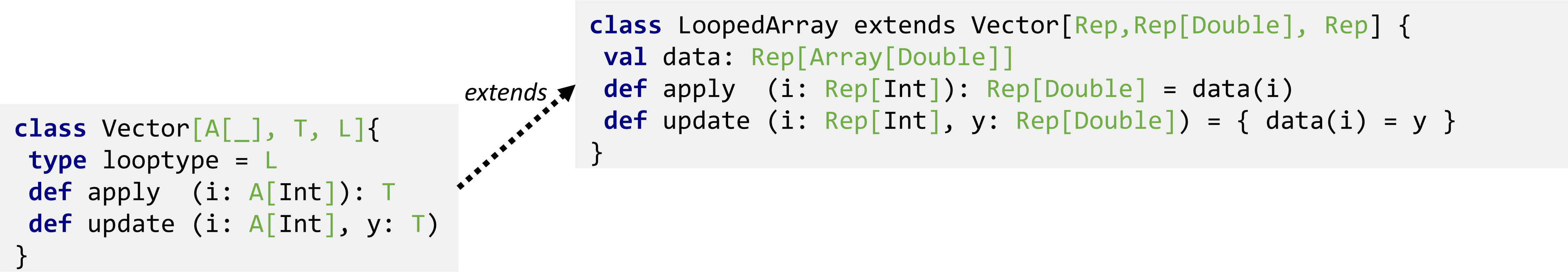
```
val a = Array[Double](3)
val b = List[Double](3)

f(a,b)
```

```
def f[T](x: DataAbstraction, y: DataAbstraction) {
 x(0) = y(0)
}
```

Abstraction over Data Layout

```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

Abstraction over Data Layout

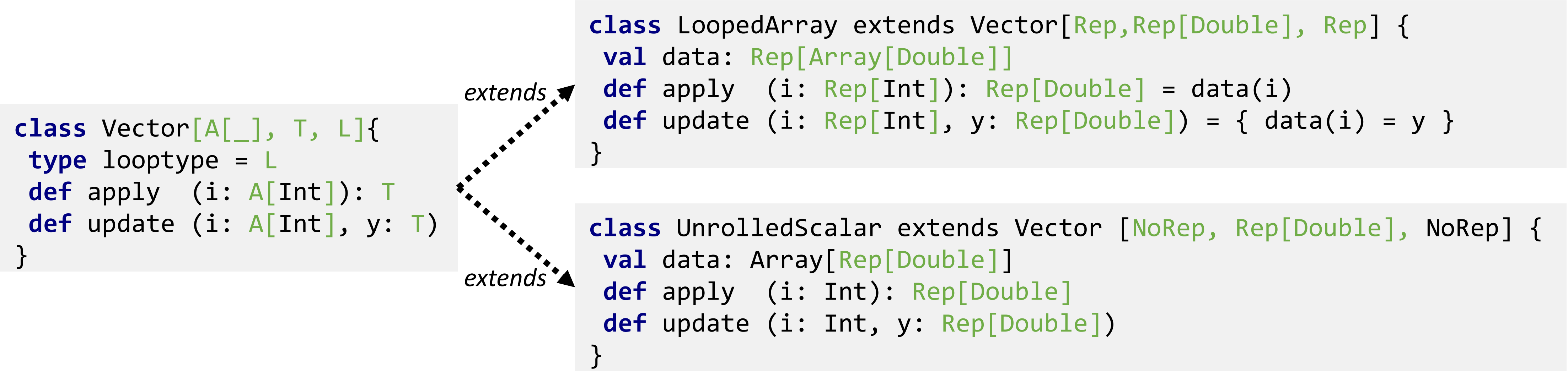


```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

extends

```
class LoopedArray extends Vector[Rep,Rep[Double], Rep] {
 val data: Rep[Array[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

Abstraction over Data Layout



```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

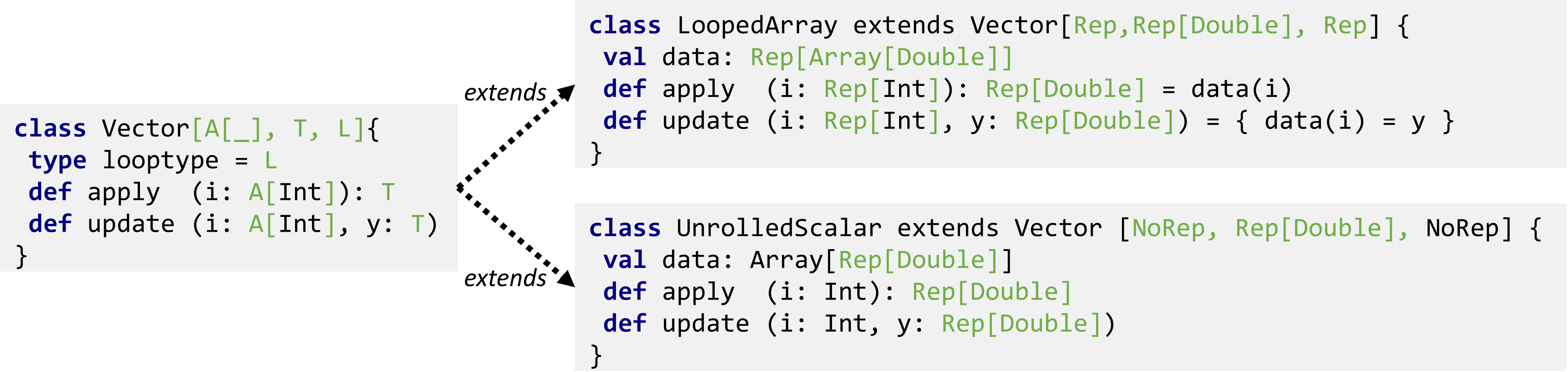
extends

```
class LoopedArray extends Vector[Rep,Rep[Double], Rep] {
 val data: Rep[Array[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

extends

```
class UnrolledScalar extends Vector [NoRep, Rep[Double], NoRep] {
 val data: Array[Rep[Double]]
 def apply  (i: Int): Rep[Double]
 def update (i: Int, y: Rep[Double])
}
```

Abstraction over Data Layout



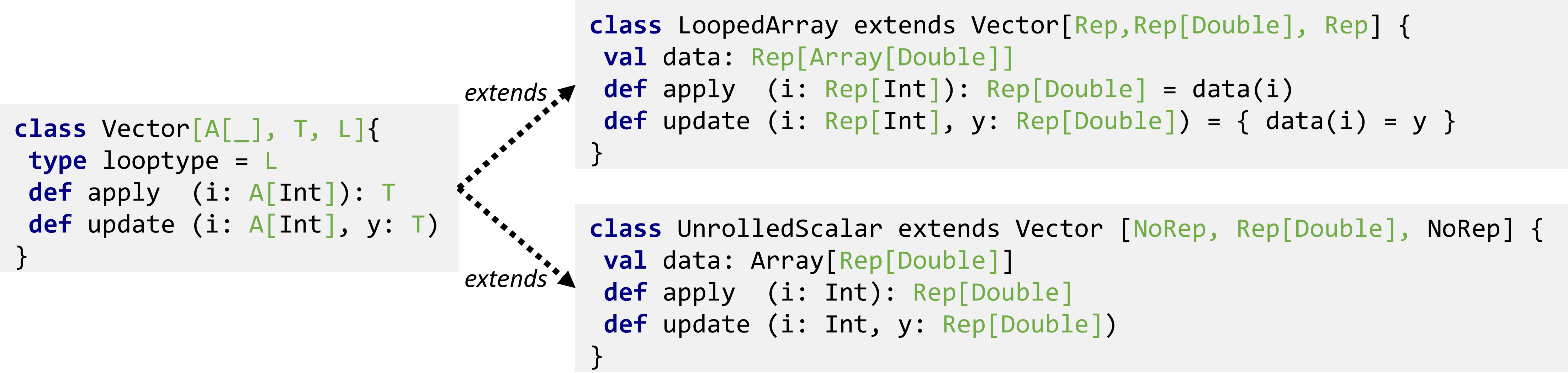
```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

```
class LoopedArray extends Vector[Rep,Rep[Double], Rep] {
 val data: Rep[Array[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

```
class UnrolledScalar extends Vector [NoRep, Rep[Double], NoRep] {
 val data: Array[Rep[Double]]
 def apply  (i: Int): Rep[Double]
 def update (i: Int, y: Rep[Double])
}
```

```
def f[A1[_], A2[_], L1, L2, T] (x: Vector[A1[_], T, L1], y: Vector[A1[_], T, L2]) ={
   for (i <- 0 until 2: y.looptype) {
     y(2*i)   = x(i*2) + x(i*2+1)
     y(2*i+1) = x(i*2) - x(i*2+1)
   }
}

val x = new LoopedArray
val y = new UnrolledScalar
f(x,y)
```

Abstraction over Data Layout



```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

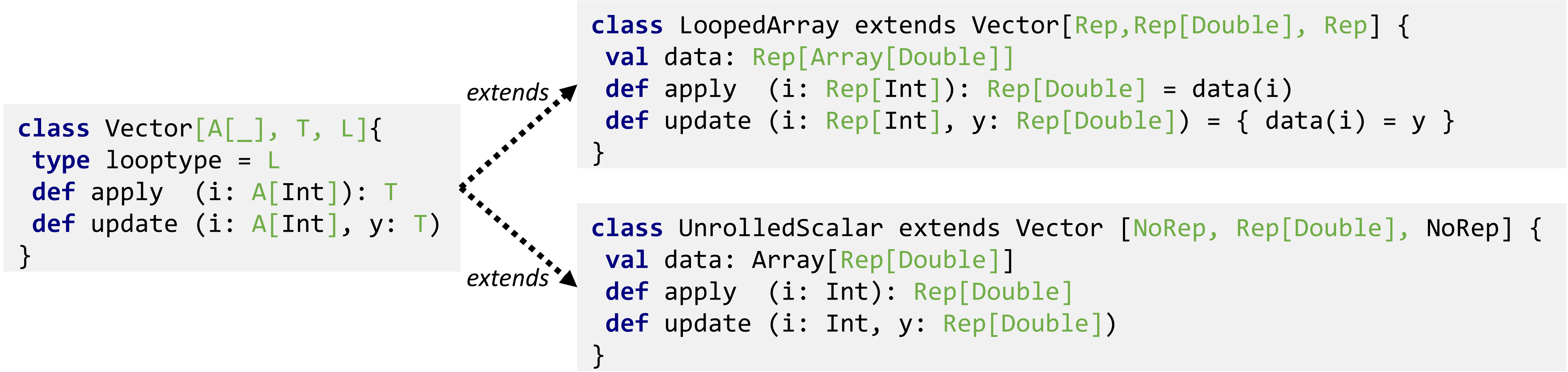
```
class LoopedArray extends Vector[Rep,Rep[Double], Rep] {
 val data: Rep[Array[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

```
class UnrolledScalar extends Vector [NoRep, Rep[Double], NoRep] {
 val data: Array[Rep[Double]]
 def apply  (i: Int): Rep[Double]
 def update (i: Int, y: Rep[Double])
}
```

```
def f[A1[_], A2[_], L1, L2, T] (x: Vector[A1[_], T, L1], y: Vector[A1[_], T, L2]) ={
   for (i <- 0 until 2: y.looptype) {
     y(2*i)   = x(i*2) + x(i*2+1)
     y(2*i+1) = x(i*2) - x(i*2+1)
   }
}

val x = new LoopedArray
val y = new UnrolledScalar
f(x,y)
```

x

Abstraction over Data Layout



```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

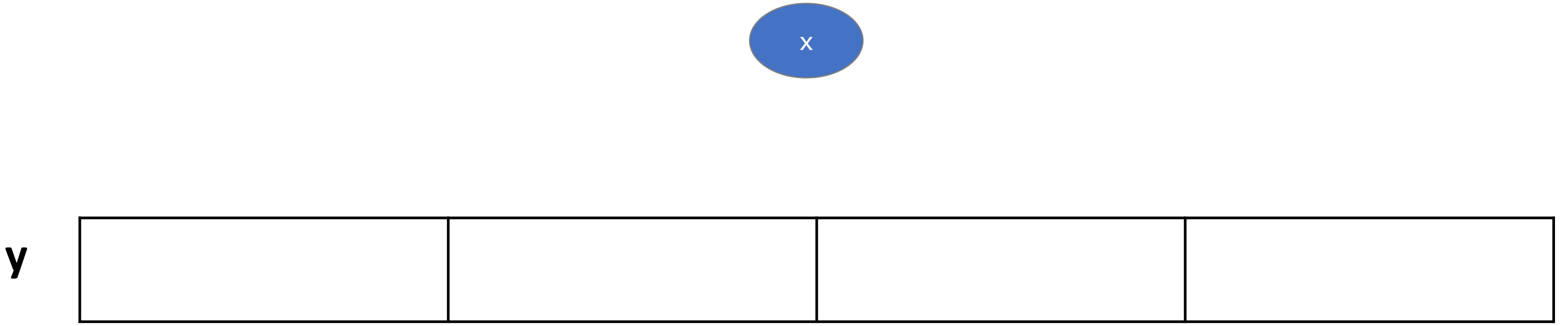
extends

```
class LoopedArray extends Vector[Rep,Rep[Double], Rep] {
 val data: Rep[Array[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

extends

```
class UnrolledScalar extends Vector [NoRep, Rep[Double], NoRep] {
 val data: Array[Rep[Double]]
 def apply  (i: Int): Rep[Double]
 def update (i: Int, y: Rep[Double])
}
```

```
def f[A1[_], A2[_], L1, L2, T] (x: Vector[A1[_], T, L1], y: Vector[A1[_], T, L2]) ={
   for (i <- 0 until 2: y.looptype) {
     y(2*i)   = x(i*2) + x(i*2+1)
     y(2*i+1) = x(i*2) - x(i*2+1)
   }
}

val x = new LoopedArray
val y = new UnrolledScalar
f(x,y)
```



Abstraction over Data Layout

```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

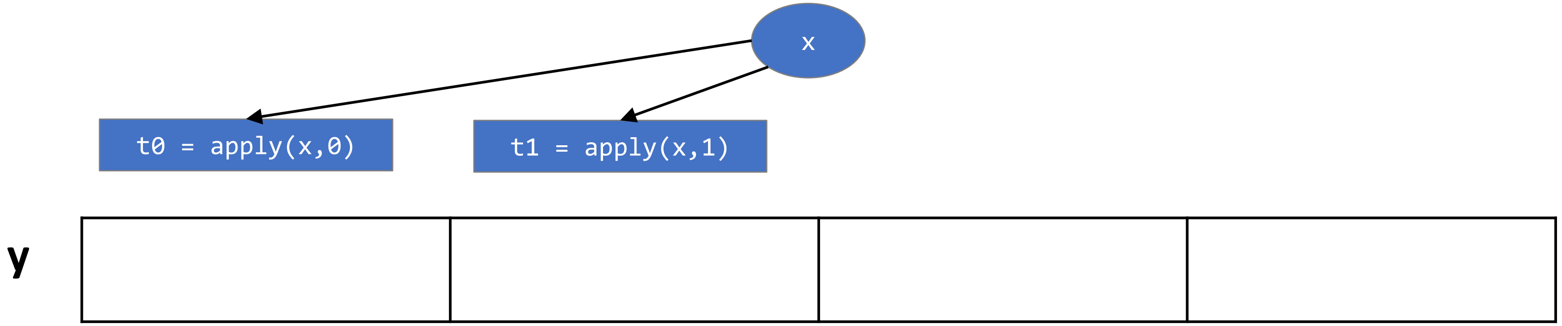
extends

```
class LoopedArray extends Vector[Rep,Rep[Double], Rep] {
 val data: Rep[Array[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

extends

```
class UnrolledScalar extends Vector [NoRep, Rep[Double], NoRep] {
 val data: Array[Rep[Double]]
 def apply  (i: Int): Rep[Double]
 def update (i: Int, y: Rep[Double])
}
```

```
def f[A1[_], A2[_], L1, L2, T] (x: Vector[A1[_], T, L1], y: Vector[A1[_], T, L2]) ={
   for (i <- 0 until 2: y.looptype) {
     y(2*i)   = x(i*2) + x(i*2+1)
     y(2*i+1) = x(i*2) - x(i*2+1)
   }
}

val x = new LoopedArray
val y = new UnrolledScalar
f(x,y)
```



Abstraction over Data Layout

```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

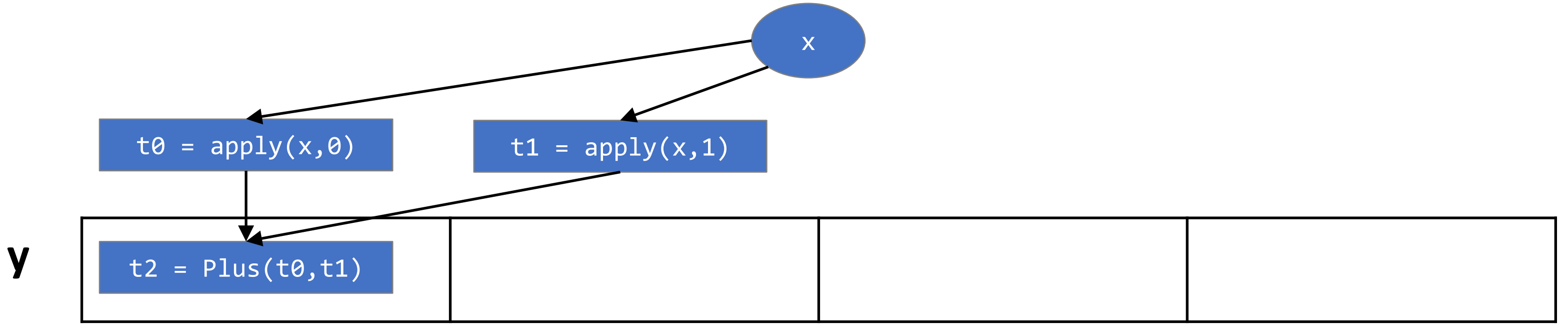
extends

```
class LoopedArray extends Vector[Rep,Rep[Double], Rep] {
 val data: Rep[Array[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

extends

```
class UnrolledScalar extends Vector [NoRep, Rep[Double], NoRep] {
 val data: Array[Rep[Double]]
 def apply  (i: Int): Rep[Double]
 def update (i: Int, y: Rep[Double])
}
```

```
def f[A1[_], A2[_], L1, L2, T] (x: Vector[A1[_], T, L1], y: Vector[A1[_], T, L2]) ={
   for (i <- 0 until 2: y.looptype) {
     y(2*i)   = x(i*2) + x(i*2+1)
     y(2*i+1) = x(i*2) - x(i*2+1)
   }
}

val x = new LoopedArray
val y = new UnrolledScalar
f(x,y)
```



Abstraction over Data Layout

```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

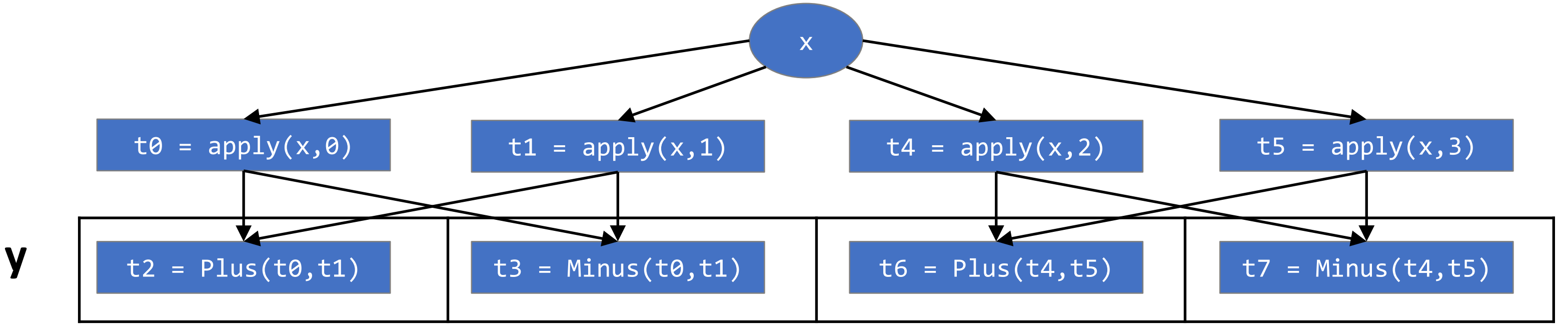
extends

```
class LoopedArray extends Vector[Rep,Rep[Double], Rep] {
 val data: Rep[Array[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

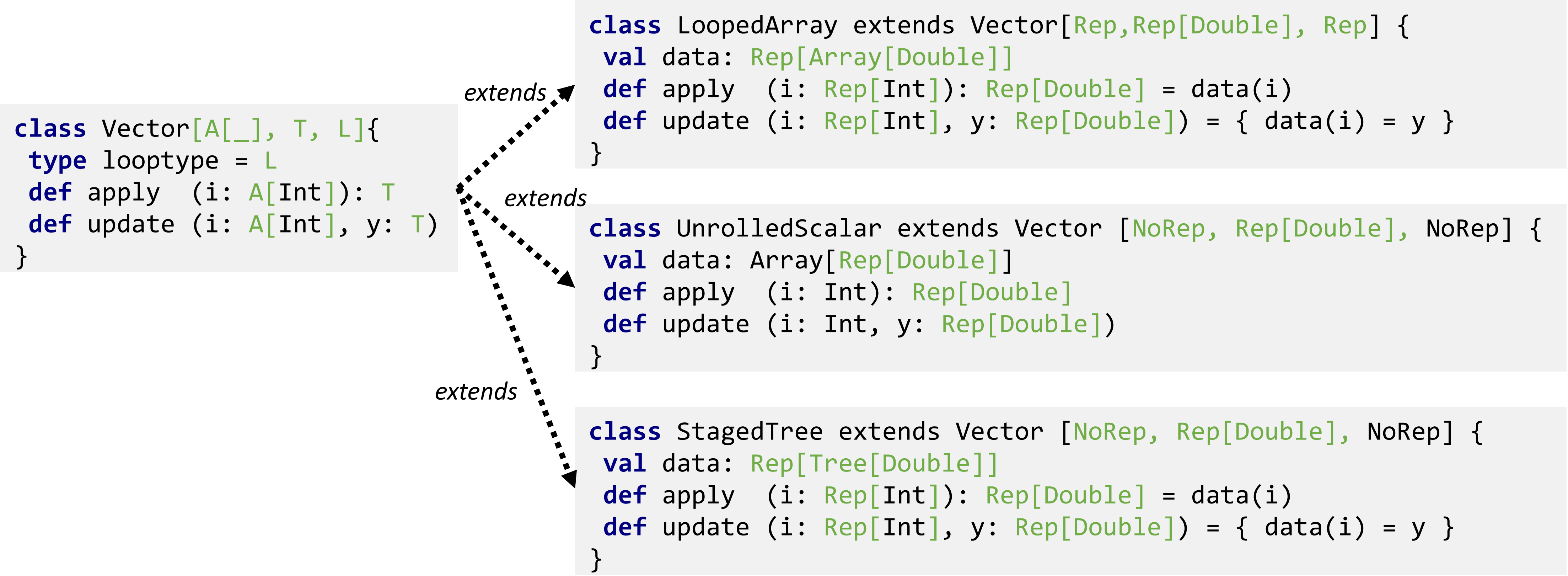
extends

```
class UnrolledScalar extends Vector [NoRep, Rep[Double], NoRep] {
 val data: Array[Rep[Double]]
 def apply  (i: Int): Rep[Double]
 def update (i: Int, y: Rep[Double])
}
```

```
def f[A1[_], A2[_], L1, L2, T] (x: Vector[A1[_], T, L1], y: Vector[A1[_], T, L2]) ={
   for (i <- 0 until 2: y.looptype) {
     y(2*i)   = x(i*2) + x(i*2+1)
     y(2*i+1) = x(i*2) - x(i*2+1)
   }
}

val x = new LoopedArray
val y = new UnrolledScalar
f(x,y)
```



Abstraction over Data Layout



```
class Vector[A[_], T, L]{
 type looptype = L
 def apply  (i: A[Int]): T
 def update (i: A[Int], y: T)
}
```

```
class LoopedArray extends Vector[Rep,Rep[Double], Rep] {
 val data: Rep[Array[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

```
class UnrolledScalar extends Vector [NoRep, Rep[Double], NoRep] {
 val data: Array[Rep[Double]]
 def apply  (i: Int): Rep[Double]
 def update (i: Int, y: Rep[Double])
}
```

```
class StagedTree extends Vector [NoRep, Rep[Double], NoRep] {
 val data: Rep[Tree[Double]]
 def apply  (i: Rep[Int]): Rep[Double] = data(i)
 def update (i: Rep[Int], y: Rep[Double]) = { data(i) = y }
}
```

Option 3: Staging in Scala

Σ-SPL

Code Style

Data Layout

AST

Code

$$T_m^n \left(\sum_{j=0}^{2} S(h_j)\, \mathbf{DFT}_2\, G(h_j) \right)$$

Looped Code

Unrolled Code

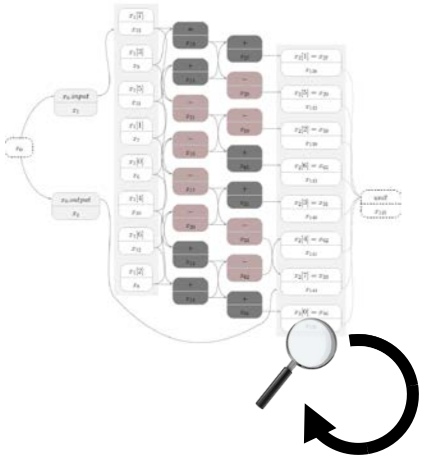
Code with Precomputation

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

C99 Complex
Interleaved Complex
Split Complex

Staging decisions



rewrite

```
def f[A1[_], A2[_], L1, L2, T]
(x: Vector[A1[_], T, L1], y: Vector[A1[_], T, L2]) …
```

Benefits

Single Decision Point for Data Layout and Code Style

```
val sigmaspl = …
val f = sigmaspl.translate
```

Creating the Meta Function

```
val input =  new LoopedArrayInterleavedComplex
val output = new LoopedArraySplitComplex
val unrolled_code = new UnrolledScalarizedC99Complex
val twiddles = TwiddleComputation[Rep]
```

Staging decisions

```
val ast = f(input,output,unrolled_code,twiddles)
```

Instantiation

Benefits

Single Decision Point for Data Layout and Code Style

```
val sigmaspl = …
val f = sigmaspl.translate
```

Creating the Meta Function

```
val input =  new LoopedArrayInterleavedComplex
val output = new LoopedArraySplitComplex
val unrolled_code = new UnrolledScalarizedC99Complex
val twiddles = TwiddleComputation[Rep]
```

Staging decisions

```
val ast = f(input,output,unrolled_code,twiddles)
```

Instantiation

Single Maintenance Point for Translations

```
def f[A1[_], A2[_], L1, L2, T] (x: Vector[A1[_], T, L1], y: Vector[A1[_], T, L2]) ={
   for (i <- 0 until 2: y.looptype) {
     y(2*i)   = x(i*2) + x(i*2+1)
     y(2*i+1) = x(i*2) - x(i*2+1)
   }
}
```

Benefits

Single Decision Point for Data Layout and Code Style

```
val sigmaspl = …
val f = sigmaspl.translate
```

Creating the Meta Function

```
val input =  new LoopedArrayInterleavedComplex
val output = new LoopedArraySplitComplex
val unrolled_code = new UnrolledScalarizedC99Complex
val twiddles = TwiddleComputation[Rep]
```

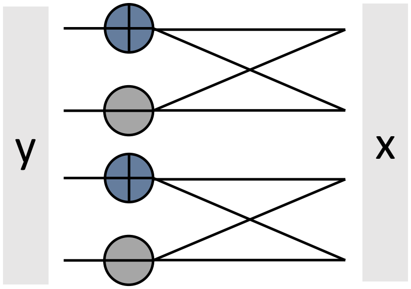
Staging decisions

```
val ast = f(input,output,unrolled_code,twiddles)
```

Instantiation

Single Maintenance Point for Translations

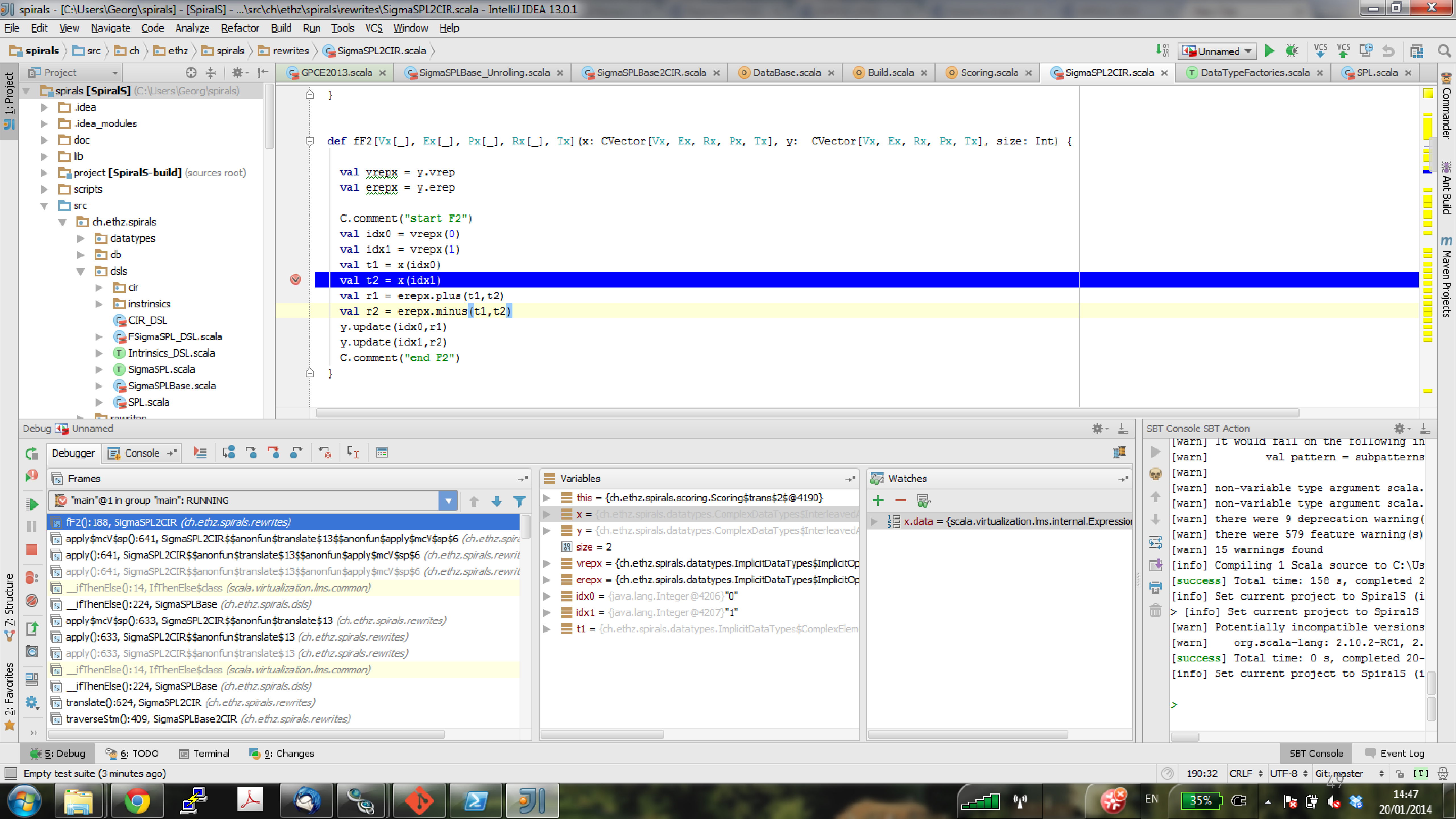
```
def f[A1[_], A2[_], L1, L2, T] (x: Vector[A1[_], T, L1], y: Vector[A1[_], T, L2]) ={
   for (i <- 0 until 2: y.looptype) {
     y(2*i)   = x(i*2) + x(i*2+1)
     y(2*i+1) = x(i*2) - x(i*2+1)
   }
}
```



Benefits

Certain Errors Manifest at Compile Time

DFT(47) has ~16.000 Decomposition



```
  }


  def fF2[Vx[_], Ex[_], Px[_], Rx[_], Tx](x: CVector[Vx, Ex, Rx, Px, Tx], y:  CVector[Vx, Ex, Rx, Px, Tx], size: Int) {

    val vrepx = y.vrep
    val erepx = y.erep

    C.comment("start F2")
    val idx0 = vrepx(0)
    val idx1 = vrepx(1)
    val t1 = x(idx0)
    val t2 = x(idx1)
    val r1 = erepx.plus(t1,t2)
    val r2 = erepx.minus(t1,t2)
    y.update(idx0,r1)
    y.update(idx1,r2)
    C.comment("end F2")
  }
```

```
"main"@1 in group "main": RUNNING
fF2():188, SigmaSPL2CIR (ch.ethz.spirals.rewrites)
apply$mcV$sp():641, SigmaSPL2CIR$$anonfun$translate$13$$anonfun$apply$mcV$sp$6 (ch.ethz.spira
apply():641, SigmaSPL2CIR$$anonfun$translate$13$$anonfun$apply$mcV$sp$6 (ch.ethz.spirals.rewrit
apply():641, SigmaSPL2CIR$$anonfun$translate$13$$anonfun$apply$mcV$sp$6 (ch.ethz.spirals.rewrit
__ifThenElse():14, IfThenElse$class (scala.virtualization.lms.common)
__ifThenElse():224, SigmaSPLBase (ch.ethz.spirals.dsls)
apply$mcV$sp():633, SigmaSPL2CIR$$anonfun$translate$13 (ch.ethz.spirals.rewrites)
apply():633, SigmaSPL2CIR$$anonfun$translate$13 (ch.ethz.spirals.rewrites)
apply():633, SigmaSPL2CIR$$anonfun$translate$13 (ch.ethz.spirals.rewrites)
__ifThenElse():14, IfThenElse$class (scala.virtualization.lms.common)
__ifThenElse():224, SigmaSPLBase (ch.ethz.spirals.dsls)
translate():624, SigmaSPL2CIR (ch.ethz.spirals.rewrites)
traverseStm():409, SigmaSPLBase2CIR (ch.ethz.spirals.rewrites)
```

```
this = {ch.ethz.spirals.scoring.Scoring$trans$2$@4190}
x = {ch.ethz.spirals.datatypes.ComplexDataTypes$InterleavedA
y = {ch.ethz.spirals.datatypes.ComplexDataTypes$InterleavedA
size = 2
vrepx = {ch.ethz.spirals.datatypes.ImplicitDataTypes$ImplicitOp
erepx = {ch.ethz.spirals.datatypes.ImplicitDataTypes$ImplicitOp
idx0 = {java.lang.Integer@4206}"0"
idx1 = {java.lang.Integer@4207}"1"
t1 = {ch.ethz.spirals.datatypes.ImplicitDataTypes$ComplexElem
```

```
x.data = {scala.virtualization.lms.internal.Expressio
```

```
[warn] It would fail on the following in
[warn]          val pattern = subpatterns
[warn]
[warn] non-variable type argument scala.
[warn] non-variable type argument scala.
[warn] there were 9 deprecation warning(
[warn] there were 579 feature warning(s)
[warn] 15 warnings found
[info] Compiling 1 Scala source to C:\Us
[success] Total time: 158 s, completed 2
[info] Set current project to SpiralS (i
> [info] Set current project to SpiralS
[warn] Potentially incompatible versions
[warn]    org.scala-lang: 2.10.2-RC1, 2.
[success] Total time: 0 s, completed 20-
[info] Set current project to SpiralS (i

>
```

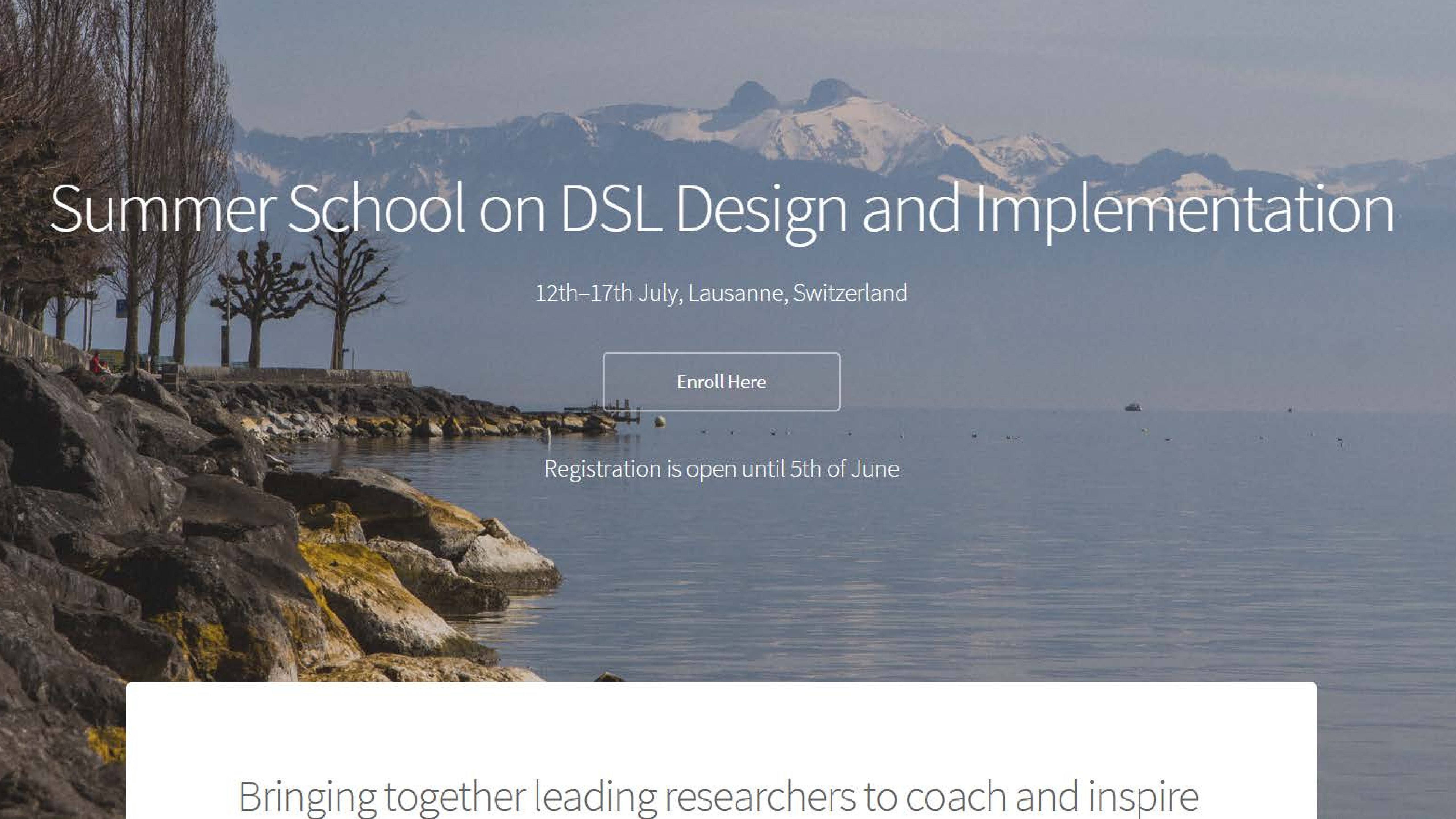
Empty test suite (3 minutes ago)

References

http://spiral.net/

http://spiral.net/software/spiral-scala.html

- Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky and Markus Püschel*. Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries, GPCE 2013*
- Tiark Rompf, Martin Odersky. *Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs.* Commun. ACM 55(6): 121-130 (2012)
- Markus Püschel, Franz Franchetti and Yevgen Voronenko
  Spiral
  in Encyclopedia of Parallel Computing, Eds. David Padua, Springer 2011



Summer School on DSL Design and Implementation

12th–17th July, Lausanne, Switzerland

Registration is open until 5th of June

Bringing together leading researchers to coach and inspire