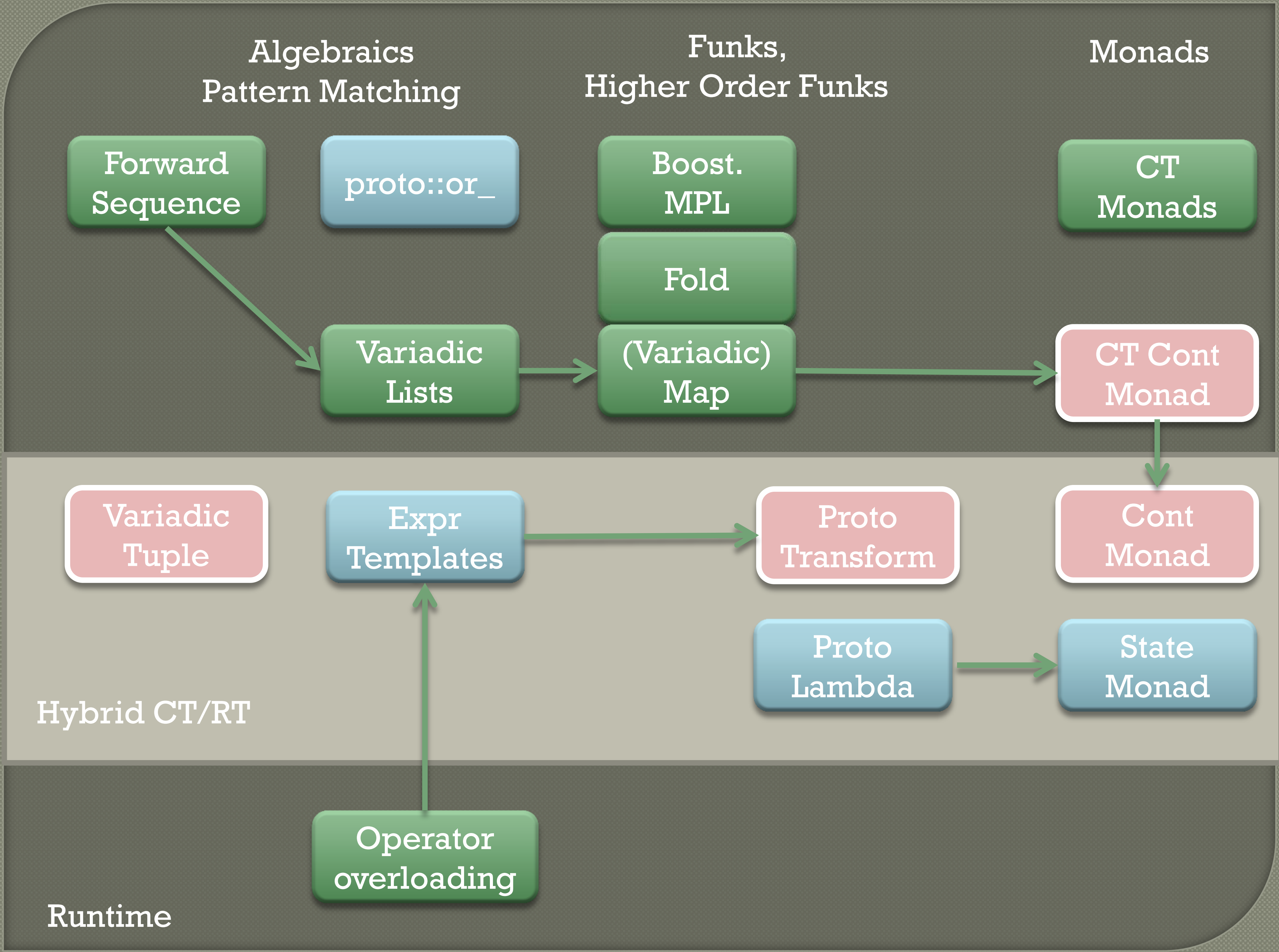
Compile-Time/Run-Time Functional Programming in C++

Bartosz Milewski
Eric Niebler


Algebraics
Pattern Matching
Funks,
Higher Order Funks
Monads
Forward Sequence
proto::or_
Boost. MPL
CT Monads
Fold
Variadic Lists
(Variadic) Map
CT Cont Monad
Variadic Tuple
Expr Templates
Proto Transform
Cont Monad
Proto Lambda
State Monad
Hybrid CT/RT
Operator overloading
Runtime

Compile-Time Continuation Monad

Or how to compose variadic templates

CT Functions

```
is_zero 0 = True
is_zero x = False
```

```
template<class T> struct
isPtr {
   static const bool value = false;
};

template<class U> struct
isPtr<U*> {
   static const bool value = true;
};
```

Variadic Templates

```
count [] = 0
count (head:tail) = 1 + count tail
```

```
template<class... list> struct count;

template<> struct
count<> {
    static const int value = 0;
};

template<class head, class... tail> struct
count<head, tail...> {
    static const int value = 1 + count<tail...>::value;
};

int n = count<int, char, long>::value;
```

Higher Order Function

```
all pred [] = True
all pred (head:tail) = (pred head) && (all pred tail)
```

```
template<template<class> class predicate, class... list> struct
all;

template<template<class> class predicate> struct
all<predicate> {
    static const bool value = true;
};

Continued…
```

Higher Order Function

```
all pred [] = True
all pred (head:tail) = (pred head) && (all pred tail)
```

```
template< template<class> class predicate,
          class head,
          class... tail>
struct
all<predicate, head, tail...> {
   static const bool value = predicate<head>::value
                          && all<predicate, tail...>::value;
};
```

Map, List Comprehension

```
map f lst = [f x | x <- lst]
```

```
// Does not compile! Can't return a template parameter pack!

template<template<class> class f, class... lst> struct
map {
    typedef typename f<lst>::type... type;
};
```

Continuation Map

```
mapCont cont f lst = cont [f x | x <- lst]
```

```
template<template<class...> class cont,
            template<class> class f,
            class... lst>
struct
mapCont {
  static const int value = cont<typename f<lst>::type ... >::value;
};
```

Composability

```
mapTwiceCont k f g xs = mapCont (\ys -> mapCont k g ys) f xs
```

```
template<template<class...> class outerCont,
            template<class> class f,
            template<class> class g,
            class... xs>
struct
mapTwiceCont {
  template<class...ys>
  struct lambda {
     static const int value = mapCont<outerCont, g, ys...>;
  }
  static const int value = mapCont<lambda, f, xs...>::value
};
```

Continuation Monad

Asynchronous Programming

Continuator

```
newtype Continuator r a = CTR ((a->r)->r)
runCont (CTR action) k = action k
```

```
template<class T, class Result>              // <- CT
struct Continuator {
    Result runCont(function<Result(T)> k); // <- RT
};
```

Continuation is not available at compile time, it is available at runtime. At compile time we compose continuators, creating bodies for runtime functions.

Continuation Monad

```
instance Monad (Continuator r) where
    return x = CTR (\k -> k x)
```

```
template<class T>
struct Return
{
    Return(T val) : _val(val) {}
    void runCont(function<void(T)> k)
    {
        k(_val);
    }
    T _val;
};
```

Return creates a continuator, which given a runtime value calls a runtime continuation with that value. (Lifting value to monad.)

```
ktor >>= rest = CTR (\k ->        -- k has type b->r
       runCont ktor (\a -> runCont (rest a) k))
```

```
template<class T, class C1, class C2>
struct Bind
{
    Bind(C1 & ktor, function<C2(T)> rest)
        : _ktor(ktor), _rest(rest)
    {}
    void runCont(function<void(T)> k)
    {
        function<C2(T)> rest = _rest;
        function<void(T)> lambda = [k, rest](T a)
        {
            return rest(a).runCont(k);
        };
        _ktor.runCont(lambda);
    }
    C1 _ktor;
    function<C2(T)> _rest;
};
```

Running Async with Continuation

```
void asyncApi(function<void(string)> handler)
{
    thread th([handler](){
        cout << "Started async\n";
        this_thread::sleep_for(chrono::seconds(3));
        handler("Done async");
    });
    th.detach();
}

// Continuator
struct AsyncApi {
    void runCont(function<void(string)> k) {
        asyncApi(k);
    }
};
```

```haskell
loop s =
   print s
   do
     s <- async
     loop s
```

```haskell
loop s =          -- desugared
   print s
   async >>= \s -> loop s
```

```cpp
struct Loop {
    Loop(string s) : _s(s) {}

    void runCont(function<void(string)> k) {
        cout << _s << endl;
        Bind<string, AsyncApi, Loop>(
            AsyncApi(),
            [](string s)
            {
                return Loop(s); // recursion?
            }).runCont(k);
    }
    string _s;
};
```

The code before the async call and after it, in one place.
Similar to C# ContinueWith. Sugar coated using “await”

```
void main(){
    Loop("Loop: ").runCont([](string s)
    {
        cout << s << endl;
    });

    for(int i = 0; i < 200; ++i)
    {
        cout << i << endl;
        this_thread::sleep_for(chrono::seconds(1));
    }
}
```

```
Loop:
0
Started async
1
2
3
Done async
Started async
4
5
6
Done async
```

Higher Order Functions

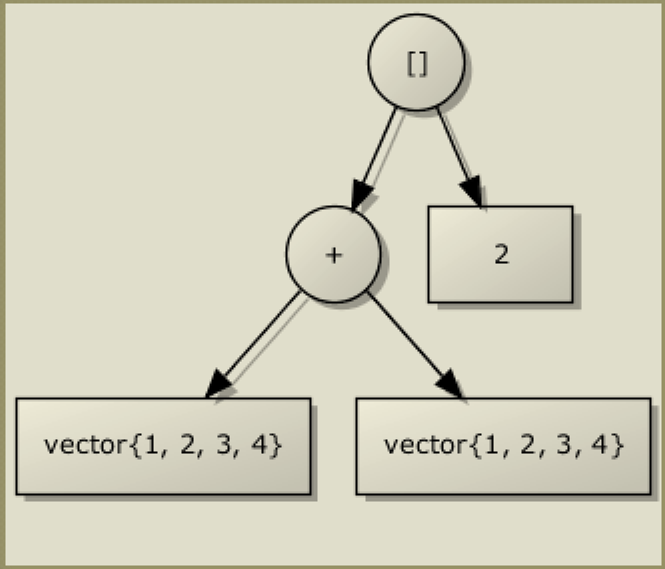
Proto Optimize and Others

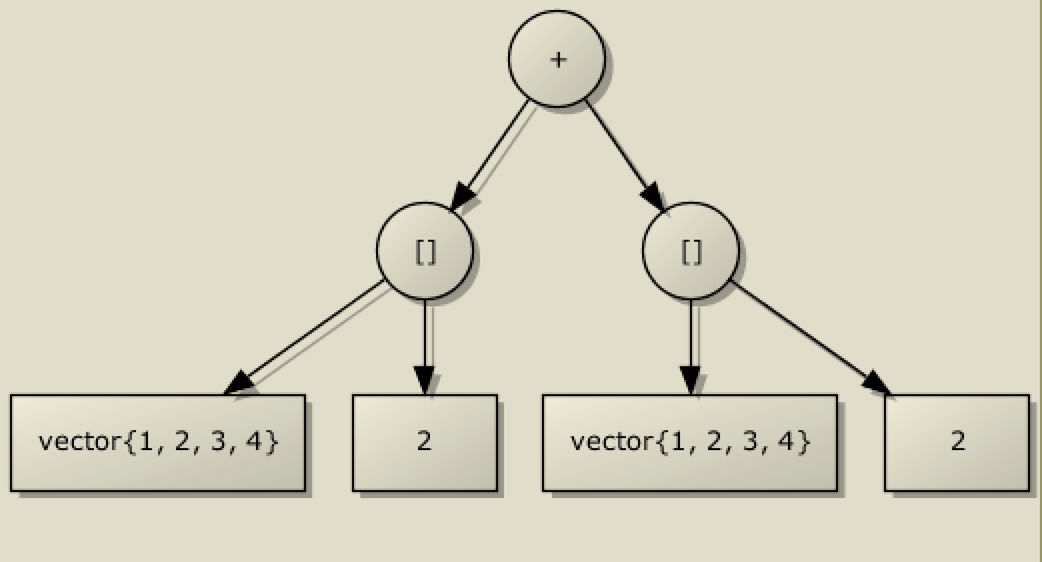
Understanding Proto
Simple Vector Expressions

```
data Expression a =
    IntE (a -> Int)
  | SubscriptE (Expression a) (Expression a)
  | PlusE (Expression a) (Expression a)
  | VectorE (a -> [Int])   -- "Runtime" function
```

```
// A vector expression consists of vector
// terminals, subscript, and addition
struct VecExpr:
  or_<
    subscript<VecExpr, terminal<_> >,
    plus<VecExpr, VecExpr>,
    terminal<_>
  >
{};
```

Proto: The Optimizing Transform





```
optimize (PlusE lftE rgtE) =
  PlusE (optimize lftE) (optimize rgtE)

optimize (SubscriptE lftE rgtE) =
  distributeIndex lftE (optimize rgtE)

optimize other = other
```

```
struct Optimize:
  or_<
    // Leave terminals alone
    terminal<_>,
    plus<Optimize, Optimize>,
    when<
      // Distribute the subscript operation over the
      // left child. Use the right child as the state.
      subscript<DistributeIndex, terminal<_> >,
      DistributeIndex(_left(_), _right(_))
    >
  >
{};
```

```haskell
distributeIndex (VectorE vf) idxEF =
  SubscriptE (VectorE vf) idxEF

distributeIndex (PlusE lftE rgtE) idxEF =
  let lftE' = distributeIndex lftE idxEF
      rgtE' = distributeIndex rgtE idxEF
  in PlusE lftE' rgtE'
```

```cpp
struct DistributeIndex:
  or_<
    when< terminal<_>,
      _make_subscript(_, _state)
    >,
    when< plus<DistributeIndex, DistributeIndex>,
      _make_plus(
        DistributeIndex( _left(_), _state ),
        DistributeIndex( _right(_), _state )
      )
    >
  >
{};
```

Understanding Transforms

- Dual nature: CT/RT
- CT transform driven by type of expression tree
- Produces a transformation of the runtime expression tree
- Examples of transforms
  - Evaluation (reduce tree to one terminal node)
  - Optimization
  - Type Check (returns the type of value produced by expression)