Annales Univ. Sci. Budapest., Sect. Comp. 37 (2012) 157–174

A C++ PEARL – SELF-REFERRING STREAMS

Attila Góbi, Zalán Szűgyi and Tamás Kozsik
(Budapest, Hungary)

Communicated by Zoltán Horváth

(Received January 15, 2012; revised March 5, 2012;
accepted March 10, 2012)

Abstract. Since C++ is a multiparadigm language, experimenting with functional programming techniques in this language seems fruitful. Self-referencing data is widely used in lazy functional languages. In the most interesting cases of self-referencing we produce infinite data. In this case it is possible to express infinite data with a finite structure. This paper makes the concept of stream-oriented programming available for the C++ programmer.

1. Introduction

Streams – as used in this paper – are infinite sequences of data. They often appear in functional programming languages, and they are completely natural in lazy languages. *Codata* [6] is a widely used term to refer to coinductively defined types, and hence infinite data structures. Streams are the simplest example of codata.

Key words and phrases: C++, stream, C++11, codata.
2010 Mathematics Subject Classification: 68N19.
1998 CR Categories and Descriptors: D.3.3
The Research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

Since C++ is striving for being a multiparadigm language, experimenting with unorthodox programming techniques in this language is very exciting. While self-referencing is an often used construct to build infinite data in lazy functional languages, iterating over infinite sequences seems to be cumbersome in an imperative language. Consider first the following Haskell definition:

```
ones = 1:ones
```

This is an infinite list of ones (note that ":" is the list constructor taking the head element and the rest of the list). This can be represented nicely in C++ with a cyclic list.

```
1  template <typename T>
2  struct list {
3      T value;
4      list<T> *next;
5  };
6  list<int> ones = {1, &ones};
```

While this simple (and useless) example is quite easy to implement in C++, more complex (and useful) examples are much harder to write down. For example, the increasing sequence of natural numbers, `nats`, is not possible to represent as a finite cyclic list, even though it is still very easy to express in Haskell:

```
nats = 0 : map (1+) nats
```

Here, `map` is a function taking a function and a (finite or infinite) list as arguments, and applies the function to every element of the list. In this case the elements of the resulting stream, `nats`, will be `0`, `1+0`, `1+1+0`, ... – e.g. the natural numbers.

There is another solution to define `nats` – one that is more in accordance with the rest of this paper.

```
nats = 0 : zipWith (+) ones nats
```

Function `zipWith` is the elementwise application of a binary function. Here "`+`" is applied on the respective elements of `ones` and `nats`.

One further advantage of using streams is that a stream memoizes an already calculated value, so taking the first one hundred elements of a stream will be much faster the second time, especially if the calculation of the stream is costly.

This paper makes the concept of stream-oriented programming [3] available for the C++ programmer. Furthermore, the presented implementation heavily

relies on some new features of C++11 [13, 14], such as user-defined literals and rvalue references [2], so it can also be viewed as a demonstration of the new features of C++ [13]. The rest of the paper is structured as follows. In Section 2 some examples are discussed and a short introduction to our library is given. Section 3 describes the internal representation of the streams used in the library. Section 4 is about the implementation details: it discusses the challenges arose during the implementation and the solutions for them. Section 5 presents the performance evaluation of the library. Section 6 concludes the paper and reviews related work as well as future research directions.

The source code of the library and some examples can be downloaded from the URL `http://kp.elte.hu/cppstreams`.

2. C++ and streams

The goal of the paper is to provide a library to handle infinite streams in a C++-ish way. Among others it means integration to existing C++ technologies such as operator overloading and iterators. While the details about the library can be found in Section 4, in this section a short introduction is given.

Let us start with `ones`. In C++ there is no "`:`" operator, so we need to choose another notation for use in our library. A right associative operator with precedence lower than that of "`+`" is preferred, as we will see later. Our choice is to use the "`<<=`" operator as it looks similar to standard C++ stream operators and it has the preferred precedence and associativity. So with the help of our library, our simple `ones` stream can be defined in the following way.

```
stream<int> ones = 1 <<= ones;
```

As a matter of fact, our library already defines constant streams as user defined literals – see Section 4.3.

An alternating sequence of zeroes and ones can be written as follows.

```
stream<int> bits = 0 <<= (1 <<= bits);
```

Note that the parentheses are optional in this expression because of the right associativity of the `<<=` operator.

```
stream<int> bits = 0 <<= 1 <<= bits;
```

Alternatively, this definition can be given using mutual recursion:

```
1   extern stream<int> bits0, bits1;
2   stream<int> bits0 = 0 <<= bits1;
3   stream<int> bits1 = 1 <<= bits0;
```

We have overloaded the "+" operator so that one can provide a definition of `nats` similar to the Haskell one in the previous section, but without `zipWith`. Basically, the overloaded "+" corresponds to the Haskell expression `zipWith (+)`. The precedence of `<<=` ensures that no parentheses are needed in definitions of this kind.

```
stream<int> nats = 0 <<= ones + nats;
```

The following table visualizes the previous definition.

	0	1	2	3	4	5	...			nats
	+	+	+	+	+	+	...			+
0	1	1	1	1	1	1	...	0	<<=	ones
=	=	=	=	=	=	=	...	=		
0	1	2	3	4	5	6	...	nats		

As a bit more complex example, we can express the Fibonacci sequence as a stream.

```
stream<int> fib = 0 <<= fib + (1 <<= fib);
```

To help understand this definition, here is its table:

	1	0	1	1	2	3	5	...			1<<=fib
	+	+	+	+	+	+	+	...			+
0	0	1	1	2	3	5	8	...	0	<<=	fib
=	=	=	=	=	=	=	=	...	=		
0	1	1	2	3	5	8	13	...	fib		

Of course, it is not possible in C++ to work with infinite structures, so we need a tricky representation, which will be explained in the next section. Also note that in order to process infinite data structures in C++ we can use iterators. This enables us to use streams with iterators just like reading elements of a container. However, the programmer must keep in mind that there is no `end()` iterator as streams are endless. With the help of iterators we can print the first 10 elements of the Fibonacci sequence.

```
1   stream<int>::iterator it = fib.begin();
2   for(int i=0; i<10; ++i)
3   {
4       std::cout<<*it<<"\n";
5       ++it;
6   }
```

3. Internal data structure

The internal representation of a stream is very similar to the graphs [16] found in graph rewriting systems, e.g. used in the implementation of the functional programming language Clean [8]. Consider our former example, `ones`. It can be represented with a graph as shown in Figure 1. Node "`<<=`" means

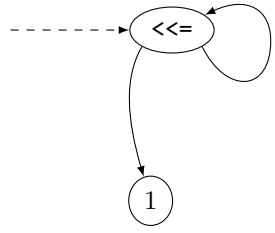
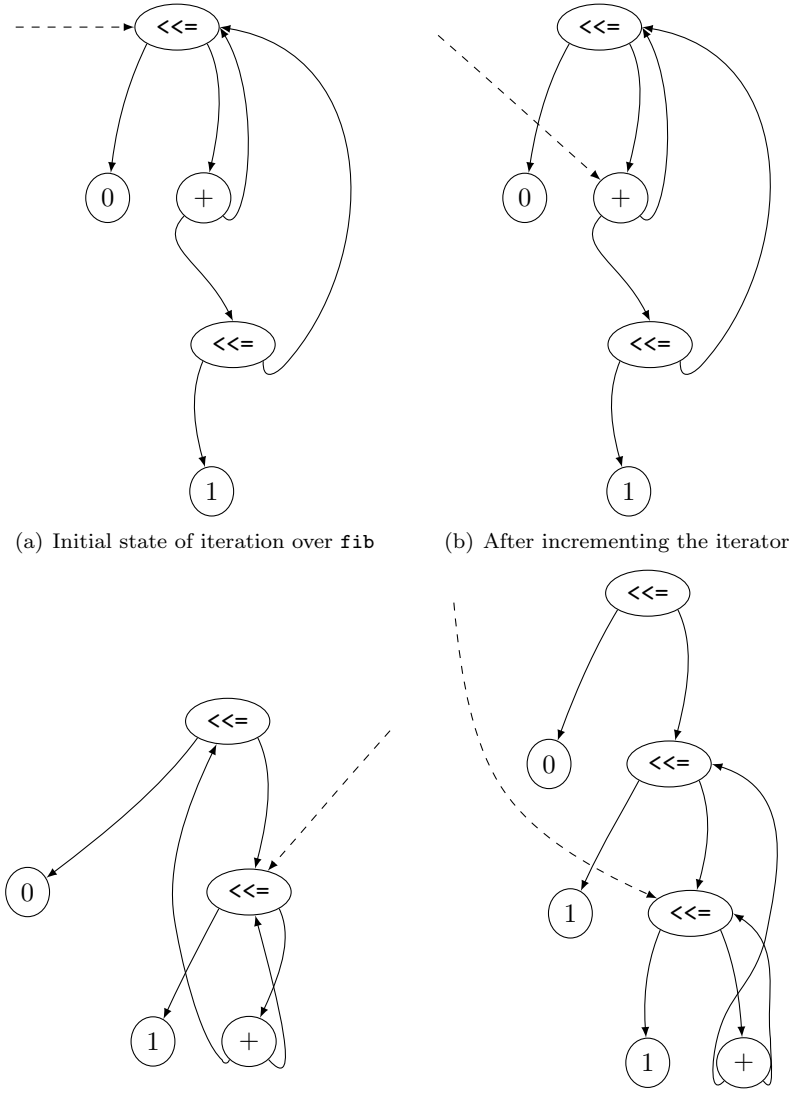


Figure 1. Ones

the stream can be decomposed into an element (head) and a remaining stream (tail). If we want to iterate over the stream, the current element (accessed by operator "`*`") will be the left successor of the node, while the remaining stream (obtainable by operator "`++`") will be the right successor. It is easy to see that in this case the value of the current element is `1`, and incrementing the iterator results in the same stream.

The Fibonacci sequence requires a "`+`" node as well (Fig. 2(a), other binary stream operators can be represented similarly). The dashed line in this figure shows where the iterator is initially pointing at. After incrementing the iterator, the iterator points at the "`+`" node (Fig. 2(b)). To increment the iterator further or to get the current value of the iterator, the "`+`" node must be evaluated first. This means rewriting the graph. The node "`+`" refers to two streams, so firstly we have to get the head values of those streams, and add them up. Then a new "`<<=`" node will be inserted with the result of the computation before the "`+`" node, and the stream references in this node will be replaced with their tails (Fig. 2(c)). Finally, Fig. 2(d) shows the graph after further incrementing the iterator and evaluating "`+`". One can also observe the effect of memoization in this figure.

An important difference between our approach and lazy functional languages is that the elements of streams would be actually calculated even if the value was not accessed (e.g. by the operator `*` of an iterator). In our case this means inserting closures to postpone the evaluation.



(a) Initial state of iteration over `fib`

(b) After incrementing the iterator

(c) After evaluating operator `+`

(d) Iterator further incremented

Figure 2. Representation of the Fibonacci sequence

4. Implementation details

The main component in our library is the class template `stream`, which represents a coinductively defined infinite sequence. The `stream` class has one template argument, which specifies the type of the elements in the sequence. It is important to note that our `stream` is different from the well-known streams of C++. In the standard library, `std::stream` is a sequence of characters, and it has an internal state holding the position (a `streambuf` object). In our case, stream is a stateless representation of an infinite sequence of a given type. The responsibility of this class is memory management, as it holds the already calculated, memoized data. State (i.e. information about current position) is held in iterators.

The core of the public interface for constructing streams consists of the operators `zipwith`, `map`, `pure` and "`<<=`" (Fig. 3).

```
1   template <typename T>
2   class stream<T>
3   {
4   public:
5     template <typename Op, typename ST1, typename ST2>
6     stream<T> zipwith(Op op, ST1 &&s1, ST2 &&s2);
7
8     template <typename Op, typename ST1> stream<T>
9     map(Op op, ST1 &&s1);
10
11    stream<T> pure(const T& v);
12    ...
13  };
14
15  template <typename ST1, typename U>
16  stream<U> operator <<= (const U& a, ST1 && s);
```

Figure 3. Public interface of streams

Operator `map` applies elementwise a unary function to a stream. Operator `zipwith` applies pairwise a binary function on the corresponding elements of two streams – the implementation of operators "`+`", "`*`" (Fig. 4) etc. is based on `zipwith`.

Operator `pure` constructs a constant stream from a value. Finally, "`<<=`" constructs a stream from a head and a tail.

```
1  template< typename ST1,
2            typename ST2,
3            typename T=typename stream_value_type<ST1>::type >
4  stream<T> operator +(ST1 &&s1, ST2 &&s2);
```

Figure 4. Signature of operators

The `ST1` and `ST2` template parameters are used to differentiate between initialized and uninitialized arguments.

4.1. Handling uninitialized data

When defining self-referring streams, references to uninitialized data occur. The only way to store uninitialized data in the representation of a stream is to store a *reference* in the respective stream node (i.e. the nodes in the graph representing streams). Furthermore, the implementation of the stream operators must make sure not to dereference uninitialized data.

Stream objects can appear as subexpressions – the temporary object storing such streams are deleted when the surrounding expression has been evaluated. Therefore, holding a reference to a temporary stream must be avoided. The stream nodes are polymorphic in this sense – they store either references (to some possibly uninitialized data) or store a copy of some temporary object. This polymorphism must be respected when streams are passed to the stream constructing operators.

One could overload the stream operators for passing arguments by value and by reference, which is standard practice in C++. The novel features C++11 [14, 13] allow for a more elegant solution: one could overload stream operators on lvalue references and rvalue references [2]. Temporary objects will be received as rvalues, possibly uninitialized streams will be received as lvalues. Note, however, that this approach still leads to code duplication. In an even better solution templates are used to generate the overloaded operators: the template parameter captures the choice between lvalue and rvalue references. In our solution this template parameter is propagated into the graph nodes: the polymorphism of the graph node is based on the choice between lvalue and rvalue references obtained when constructing the node. The storage type of a node is determined by a template metaprogram [15, 1], shown in Fig. 5. Therefore, the storage type of a node can be deduced in compile time – which is essential in our case. The necessary storage information is stored in the template parameter of the concrete `impl` classes, and no additional runtime overhead emerges. For efficiency, when a temporary stream object is to be stored, move semantics will be applied: the content of the temporary object will be moved into the new graph node, and the empty temporary object will then be deleted.

```
1   template<typename T>
2   struct storage_type
3   {
4     typedef typename std::conditional<
5           std::is_lvalue_reference<
6               typename std::remove_const<T>::type>::value,
7
8           typename std::remove_const<T>::type,
9
10          typename std::remove_reference<
11              typename std::remove_const<T>::type>::type
12    >::type type;
13  };
```

Figure 5. Template metaprogram to infer the storage type of nodes

The template metaprogram, Fig. 5 checks whether the type is `lvalue` or `rvalue` reference. If it is an lvalue reference (the stream is not temporary), the metaprogram returns with the reference type of the stream. This way initialization means reference initialization, which does not copy the element. If the type of a stream is rvalue reference, the stream is temporary. In this case the metaprogram returns with the pure type of the stream. Initializing a pure type means copying, thus the temporary object is saved.

4.2. Graph nodes

Class `stream` contains a pointer to an implementation hierarchy, which is a representation of the graph of the stream. This hierarchy is built from different implementation classes, which are all derived from the abstract class `impl`. This latter class prescribes the interface of the implementation services, and its concrete subclasses provide the implementation of the different stream constructing operators. The interface of the implementation services is as follows.

- `virtual` destructor: provides the proper destructor invocation for child classes.
- pure `virtual const T &get(const iterator&)`: returns the current element of the stream.
- pure `virtual void next(iterator&)`: jumps to the next element of the stream.

The destructor is responsible for proper deallocation of the node, `get` returns the head element of the node, while `next` moves the passed iterator to the tail. For the first sight it might be seen unnecessary to pass the current iterator to the implementation in the method `get`, but in fact it is important and will be explained later in this subsection.

4.2.1. The `addimpl` class

Class `addimpl` (Fig. 6) implements the behaviour of `operator<<=`, which extends a stream with an element as the head of the resulting stream. Technically, class `addimpl` is a pair: the first member is the inserted element, and the second one stores the implementation of the rest of the stream (in a way described in Sec. 4.1).

```
1  template<typename ST>
2  struct addimpl: public impl {
3      addimpl(const T &a, ST &&s)
4          : a_(a), s_(std::forward<ST>(s))
5      { }
6      ...
7  private:
8      const T a_;
9      typename storage_type<ST>::type s_;
10 };
```

Figure 6. `addimpl` class

4.2.2. The `zipimpl` and `mapimpl` classes

Class `mapimpl` implements an elementwise operation over the stream, and class `zipimpl` implements the binary pairwise operations over two streams (e.g. addition, subtraction...). Basically, `mapimpl` is a pair of a functor and a stream (while `zipimpl` is a triple of a functor and two streams). The basic idea is to (1) take an iterator from the contained stream; (2) execute the operator; (3) create an `addimpl` instance from the calculated value as head and the current object as tail; (4) replace the actual object with the newly created one.

Creating the iterator can be tricky as during the instantiation of the node the referred stream might not be initialized. So we have to do it when the first `get` or `next` call occurs. To avoid the overhead of checking whether the iterator is initialized we decided to split the implementation into two parts – `mapimpl` is only responsible for creating the iterators (Fig. 7) and the rest is done by `mapimpl2` (Fig. 8). After creating the iterators a `mapimpl` object deletes itself.

```
1   const T &get(const iterator &it)
2   {
3       *(it.impl_) =
4           new mapimpl2<Op, ST>(op_, std::forward<ST>(s_));
5       delete this;
6       return *it;
7   }
```

Figure 7. `get` method of `mapimpl` class

This is the reason why it is important to pass the current iterator to the `get` method of the nodes. Also note that the `impl_` field of class iterator must be mutable to make updating possible even if the iterator is constant. Returning `*it` means calling the `get` method of the newly created `mapimpl2` instance, which is defined as follows.

```
1   const T &get(const iterator &it)
2   {
3       *(it.impl_) =
4           new addimpl<stream<T>&&>(op_(*it1), stream<T>(this));
5       ++it1;
6       return *it;
7   }
```

Figure 8. `get` method of `mapimpl2` class

Here `*it` is returned again, which now means calling the method `get` of the `addimpl` class. As a side effect this makes it possible to return a reference to the computed value.

4.3. Stream literals

Simple streams such as streams containing infinitely many of the same element are often used to construct more complex streams. To make our library easier to use, we provide a smart construction to define these streams. The solution is based on the user-defined literals feature of C++11. To create a constant stream with a given element it is enough to write the element and add the `_s` postfix after it. Thus the two stream definitions below are creating the same stream.

```
1   stream<int> ones  = 1 <<= ones;
2   stream<int> also_ones = 1_s;
```

The code snippet below presents how our library creates an infinite constant stream of a given element.

```
1  stream_proxy operator "" _s (unsigned long long i)
2  {
3      return stream_proxy(i);
4  }
```

When a compiler finds a *_s* postfix it invokes the `operator""_s` of our library, where the prefix is passed as argument. Then we create a stream proxy, which can be converted into a proper stream. The source of that proxy can be seen below:

```
1  struct stream_proxy
2  {
3      stream_proxy(unsigned long long i): x(i) {}
4      template <typename T>
5      operator stream<T> ()
6      {
7          stream<T> a = static_cast<T>(x)<<=a;
8          return a;
9      }
10     unsigned long long x;
11 };
```

Figure 9. Proxy class to convert literal to stream

5. Performance

As a performance evaluation we implemented the problem ,,Number of ways of making change for n cents using coins of 1, 2, 5, 10, 20, 50 cents" [10] which is also known as Sloane's sequence A001313 [10]. This dynamic programming problem is especially applicable as a test, because it can utilize memoization in an efficient implementation. A C++ implementation without streams can be seen in Fig. 10. This solution uses a static unordered map to memoize already calculated return values. In former C++ versions unordered map was not available, so a solution using tree map has also been written.

The stream implementation (Fig. 11) utilizes memoization as well, but this is all implicit, moreover the code does not need to handle special cases. Therefore, the implementation is much shorter and more readable.

The remaining question is whether the performance of the stream implementation can be compared to the conventional ones? The answer is yes, however

```
long changedyn(int n, int max=4) {
    if(n<0) return 0;
    if(max<0) return 1;

    static std::unordered_map<int,long> cache[5];
    std::unordered_map<int,long>::iterator it =
        cache[max].find(n);

    if(it != cache[max].end())
        return cache[max][n];

    long ret = 1;
    switch(max) {
        case 4: ret += changedyn(n-50, 4);
        case 3: ret += changedyn(n-20, 3);
        case 2: ret += changedyn(n-10, 2);
        case 1: ret += changedyn(n-5, 1);
        case 0: ret += changedyn(n-2, 0);
    }
    cache[max][n] = ret;
    return ret;
}
```

Figure 10. Solution of the money changing problem without streams

```
template<typename ST>
stream<long> times(long n, long val, ST && s)
{
    if(n==0) return s;
    return times(n-1, val, val<<=std::forward<ST>(s));
}

stream<long> change1  = 1l<<=change1;
stream<long> change2  = times(2, 0, change2) + change1;
stream<long> change5  = times(5, 0, change5) + change2;
stream<long> change10 = times(10, 0, change10) + change5;
stream<long> change20 = times(20, 0, change20) + change10;
stream<long> change50 = times(50, 0, change50) + change20;
```

Figure 11. Solution of the money changing problem with streams

the average execution time is about 50% higher than the implementation with unordered list (see Fig. 12), but it is still much faster than the implementation with tree map. The same can be seen for other parameters: if the parameter is big enough (about 1000), the execution time of the stream implementation is between those of the two others (Fig 13). This is an excellent result for a proof-of-the-concept implementation.

A : Intel E7530 (1.87GHz), 32GB RAM, Ubuntu 11.10

B : Intel E8200 (2.66GHz), 4GB RAM, Ubuntu 11.04

C : Intel Q8400 (2.66GHz), 4GB RAM, Ubuntu 10.04

	Tree Map	Unordered Map	Stream	Overhead
A g++ 4.4	137.85 ± 2.27	51.28 ± 1.24	85.39 ± 4.89	66%
g++ 4.5	121.95 ± 2.12	57.41 ± 1.78	83.16 ± 2.10	45%
g++ 4.6	125.24 ± 1.22	53.86 ± 1.92	82.56 ± 2.31	53%
B g++ 4.4	121.83 ± 13.76	48.04 ± 0.68	72.42 ± 8.94	51%
g++ 4.5	113.56 ± 4.96	55.08 ± 1.97	72.15 ± 8.34	31%
C g++ 4.3	102.94 ± 7.60	41.05 ± 0.09	63.74 ± 2.95	55%
g++ 4.4	100.68 ± 4.15	42.49 ± 2.52	64.54 ± 4.15	52%

Figure 12. Average execution times of 100 runs with parameter 182000 in milliseconds

The measurement was performed by calling `gettimeofday` around the function call (in the case of the conventional implementation) or around the following code.

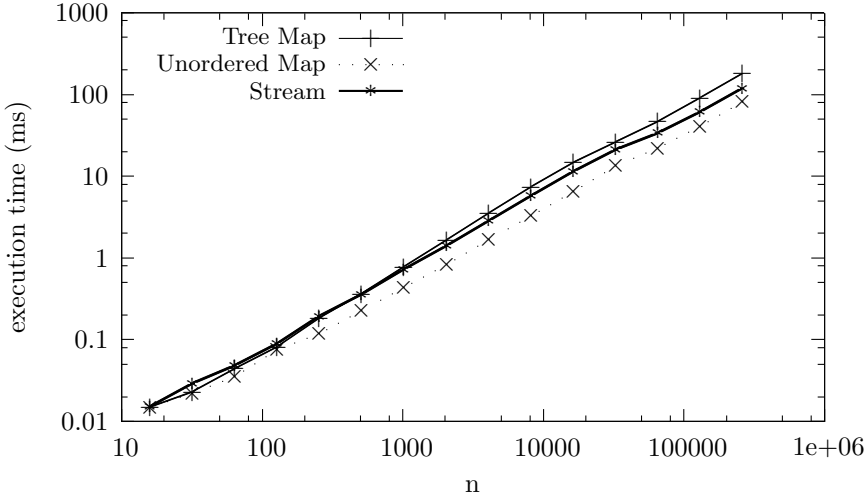
```
1  stream<long>::iterator it = change50.begin();
2  for(int i=0; i<n; ++i) ++it;
3  *it;
```

All tests have been done on 64 bit Linux platforms, the exact platforms and compiler versions are stated in the figures.

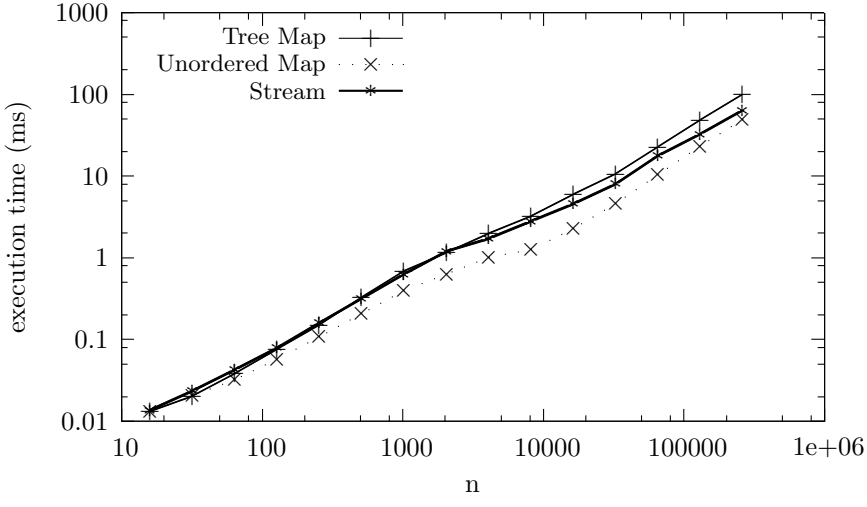
6. Conclusion

This paper presents a proof-of-the-concept implementation of the simplest kind of codata, streams, in the new standard of C++. Streams are infinite sequences of data, often with self-referring definition. Programming with streams is an elegant way to describe certain computations in a declarative style. This technique is especially useful when expressing dynamic programming problems.

The lack of lazy evaluation in C++ is the major obstacle in the implementation of codata. In our approach streams are represented by a graph



(a) Intel E7530 (1.87GHz), 32GB RAM, g++ 4.6, Ubuntu 11.10



(b) Intel M620 (2.67GHz), 2GB RAM, g++ 4.6, Ubuntu 11.10

Figure 13. Execution times for different parameters

rewriting system, allowing certain amount of laziness. Our solution uses template metaprogramming to infer the storage type of successors of graph nodes. An important contribution of this paper is the application of the novel features of C++ appeared in the C++11 standard in the following ways.

- Allowing coinductive definitions where expressions are working with partially initialized data.
- Eliminating code duplication by introducing polymorphism over lvalue and rvalue references in the implementation of the stream constructing operators.
- Improving stream syntax with user defined literals.
- Showing a case where move semantics is used not merely for the conventional, efficiency reason, but because it is exactly the right semantics of object passing to make the presented implementation feasible.

6.1. Related work

The concept of stream calculus was described by Rutten [9]. Our work was inspired by the seminal paper of Ralf Hinze [3] about using streams for programming and proving in the Haskell language.

In Lazy functional languages such as Haskell, there is nothing special in working with infinite data. Internally Haskell works by returning a closure [4] containing the necessary information on how to continue the computation, while Clean [8] uses a graph reduction similar to our approach.

Infinite input iterators [7] provide a way to handle infinite sequences of data, however, they are not capable of describing self-referring data. Another approach is to embed functional languages to metaprograms like in [11]. Unlike C++ in imperative languages supporting coroutines (such as CLU, Python, Ruby) the implementation can be quite straightforward. In Python functions over streams can use the `yield` keyword to implement an iterator [12]. The `yield` ,,returns" the next value of the iterator, and when the next element is queried, the method continues from the executed `yield` statement.

It is obvious that generating and transforming a stream is very similar to the producer-consumer problem. Using threads instead of coroutines is a quite natural solution, however the overhead can be huge, especially when a lot of agents are communicating at the same time. However, this can also be an advantage in a multi-core or in a distributed system. In [17] a functional language called D-Clean is modelled using C++, however, in this approach streams are a means of communication between threads or processes.

6.2. Future work

Being a proof-of-the-concept implementation, our streams have a number of limitations. First of all, evaluating too much elements in a stream currently causes segmentation fault during destruction. The reason for this is that, due

to the logic of the virtual destructors, the destruction of the graph is recursive, and this may cause a stack overflow.

Another important limitation comes from memoization. The current implementation does not allow the deallocation of memoized data. The best solution would be using garbage collection, or at least allowing to set explicit limits of memoized data.

One performance bottleneck is the excessive amount of memory allocation. A possible solution is to contract the memoized data, e.g. not to link the elements one by one, but using an array instead (like in tries). This solution would help to solve our first problem, but complicates a possible garbage collector and the rewriting rules. It can also help to solve the following.

The elements of the stream can only be accessed sequentially, which is natural for a stream, but makes memoization less effective – even if the value has already been calculated, we cannot access it directly. In the money change problem, a second calculation with the same parameter would not be much faster then the first one, unlike the implementations using map.

Currently the possibilities of writing a function returning a stream is very limited. Basically, anything can be implemented by creating a new class derived from `impl`, however we saw that this is complex and error-prone. It is an interesting future research task to find a more straightforward solution to enable extending the currently built-in set of operations of `stream`.

References

[1] **Abrahams, D. and A. Gurtovoy,** *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond,* Addison-Wesley (2004).

[2] **Hinnant, H.E., B. Stroustrup and B. Kozicki,** A Brief Introduction to Rvalue References `http://www.artima.com/cppsource/rvalue.html`

[3] **Hinze, R.,** Reasoning about codata, in: Z. Horváth, R. Plasmeijer and V. Zsók (Eds.) *Central European Functional Programming School Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures* Lecture Notes in Computer Science **6299**, Springer, Berlin - Heidelberg, 2010, 42–93. ISBN 978-3-642-17684-5, DOI 10.1007/978-3-642-17685-2_3

[4] **Jones, S.L.P.,** Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine, *Journal of Functional Programming*, **2**, Cambridge University Press, (1992), 127–202, DOI 10.1017/S0956796800000319

[5] **Josuttis, N.M.,** *The C++ standard library: a tutorial and reference*, Addison-Wesley (1999).

[6] **Kieburtz, R.B.,** *Codata and Comonads in Haskell* (1999)

[7] **Kozsik, T., N. Pataki, and Z. Szűgyi,** C++ Standard Template Library by infinite iterators, *Annales Mathematicae et Informaticae* **38** (2011), 75–86.

[8] **Plasmeijer, R. and M. van Eekelen,** *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley, 1993, ISBN 0-201-41663-8.

[9] **Rutten, J.,** A coinductive calculus of streams, *Math. Struct. in Comp. Science*, **15**, (2005), 93–147.

[10] **Pólya, G. and G. Szegő,** *Problems and Theorems in Analysis*, Springer-Verlag NY, 1972, **1**, 1, ISBN 978-3-540-63640-3.

[11] **Porkoláb, Z.,** Functional Programming with C++ Template Metaprograms in: Z. Horváth, R. Plasmeijer and V. Zsók (Eds.) *Central European Functional Programming School Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures* Lecture Notes in Computer Science **6299**, Springer, Berlin - Heidelberg, 2010, 306–353. ISBN 978-3-642-17684-5, DOI 10.1007/978-3-642-17685-2_9

[12] **Schemenaueri, N., T. Peters and M.L. Hetland,** PEP 255 – Simple Generators, `http://www.python.org/dev/peps/pep-0255/`

[13] *Standard for Programming Language C++*, Doc No. 3290: ISO/ISC DTR 19769 (5 April 2011) FDIS.

[14] **Stroustrup, B.,** The Design of C++0x Reinforcing C++'s proven strengths, *CC Users Journal*, **2**, (2005), 1–5.

[15] **Unruh, E.,** Prime number computation, *ANSI X3J16-94-0075/ISO WG21-462* (1994).

[16] **Wadsworth, C.P.,** *Semantics and pragmatics of the lambda calculus*, Ph.D. Thesis, Programming Research Group, Oxford University, (1971).

[17] **Zsók, V. and Z. Porkoláb,** The Distributed D-Clean Model Revisited by Templates, in: *Int'l Conf. on Numerical Analysis and Applied Mathematics*, (Chalkidiki, Greece, 2011), American Institute of Physics, 2011, 877–880. DOI 10.1063/1.3636873

A. Góbi, Z. Szűgyi and T. Kozsik
Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
{gobi,lupin,kto}@elte.hu