Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 279 (3) (2011) 27–40

www.elsevier.com/locate/entcs

Nested Lambda Expressions with Let Expressions in C++ Template Metaprograms

Ábel Sinkovics

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
e-mail: abel@elte.hu

Abstract

More and more C++ applications use template metaprograms directly or indirectly by using libraries based on them. Since C++ template metaprograms follow the functional paradigm, the well known and widely used tools of functional programming should be available for developers of C++ template metaprograms as well. Many functional languages support let expressions to bind expressions to names locally. It simplifies the source code, reduces duplications and avoids the pollution of the namespace or namespaces. In this paper we present how let expressions can be introduced in C++ template metaprograms. We also show how let expressions can be used to implement lambda expressions. The Boost Metaprogramming Library provides lambda expressions for template metaprograms, we show their limitations with nested lambda expressions and how our implementation can handle those cases as well.

Keywords: C++, boost::mpl, template metaprogramming, functional programming

1 Introduction

Let expressions are common tools in functional programming languages. Their purpose is giving names to expressions in the scope of another expression. For example in Haskell [19] let expressions look like the following:

```
let { d1 ; ... ; dn } in e
```

where `d1`, ..., `dn` are declarations in the scope of the `e` expression.

Templates were designed to capture commonalities of abstractions without performance penalties at runtime, however in 1994 Erwin Unruh showed how they can force any standard C++ compiler to execute specific algorithms as a side effect of the compilation process. This application of templates is called C++ template metaprogramming, which turned out to form a Turing-complete sub-language of C++. [3]

Template metaprogramming has many application areas today, like implementing *expression templates* [15], *static interface checking* [6,9], *active libraries* [16], or *domain specific language embedding* [4,17,8,18].

C++ template metaprograms are functional programs [7,11]. Unfortunately they have a complex syntax leading to programs that are difficult to write and understand. Template metaprograms consist of *template metafunctions* [1]. There may be sub expressions that are used multiple times in the body of a metafunction, which has to be copied leading to maintenance issues or moved to a new metafunction leading to namespace pollution. The ability to bind expressions to names locally in metafunctions could simplify both the development and maintenance of C++ template metaprograms.

Many programming languages provide tools to create no-name function objects inside an expression. These tools are called lambda expressions and by using them programmers don't have to create small utility functions when they need function objects with simple implementations. There is no lambda expression support in the current C++ standard, however there are workarounds implemented as a library. The Phoenix and Lambda libraries of Boost provide tools to build lambda expressions [17]. The upcoming standard, C++0x [13] has language support for lambda expressions. Without lambda expressions, the business logic is scattered across lots of small utility functions making the code difficult to understand and change.

The Boost Metaprogramming Library [17] provides tools to build lambda expressions for algorithms executed at compilation time. Arguments of the lambda expressions are called _1, _2, etc. This causes issues when programmers have to create nested lambda expressions inside other lambda expressions. The solution we present for let expressions can be used to implement lambda expressions in C++ template metaprograms that can express nested lambda expressions correctly. A library implemented based on the ideas presented here is available at [18].

The rest of the paper is organised as the following. In section 2 we detail the concept of let expressions. In section 3 we present our approach to add let expressions to a functional language, that has no built-in support for it. In section 4 we present how our approach can be used to implement nested lambda expressions. In section 5 we extend our approach to support recursive let expressions. We present future works in section 6 and we summarise our results in section 7.

2 Let expressions

Let expressions in Haskell bind declarations to names. A declaration can use pattern matching to bind values to names, for example:

```
let
  a = f 11
  (b, c) = returnTuple 13
in
  a + b + c
```

The above example binds the expression `f 11` to `a`. It evaluates `returnTuple 13` and tries matching the result of it to the pattern `(b, c)`. When it doesn't match,