Proceedings of the 9th International Conference on Applied Informatics
Eger, Hungary, January 29–February 1, 2014. Vol. 2. pp. 85–92
doi: 10.14794/ICAI.9.2014.2.85

Modeling C preprocessor metaprograms using purely functional languages

Máté Karácsony

Department of Programming Languages and Compilers,
Faculty of Informatics, Eötvös Loránd University, Budapest
k_mate@inf.elte.hu

Abstract

The preprocessor of the C language provides a convenient base to define relatively complex source-to-source transformations. However, writing and understanding these macros is usually fairly difficult. Lack of typing, statelessness, and uncommon syntax are the main reasons of this difficulty.

To support the development of preprocessor metaprograms, a model for each of the macros can be implemented with a function in a purely functional language. If only a well-defined, restricted set of the language is used, then a working functional implementation can be translated back into a preprocessor metaprogram. This translation leads to easily recognizable patterns in the resulting code, thus makes its maintenance and understanding easier. Furthermore, testing and prototyping may need less effort using an initial implementation in a functional language.

Keywords: Preprocessor metaprogramming, functional programming

MSC: 68N15 Programming languages

1. Introduction

With the preprocessor macros we can express advanced metaprograms. These can be used to describe source-to-source transformations to extend the capabilities of the host language itself, for example with serialization or compile-time reflection. Another possible usage of these metaprograms is calculation of complex static data based on the actual compilation options.

While most C developers are familiar with the macro language, its structure and semantics is generally very different from most imperative languages, thus some of these differences are making it difficult to understand and develop metaprograms. During macro expansions, the preprocessor does not allow to access any global, mutable state information. When every macro has a single definition which does

not change during preprocessing[1], referential transparency of macro invocations is ensured. These properties are making the macro system very similar to a purely functional language.

Since the preprocessor manipulates token streams, macros can be considered typeless. This shortcoming can easily lead to mistakes when metaprograms containing nested invocation of function-like macros are modified, especially when these macros are simulating data structures. Only a very few number of semantic checks are done on the metaprograms during processing[2]. Although expansions never directly result in failures, resulting source code is not guaranteed to be free of syntactic or semantic errors. Because the locations of these errors are often pointing into the preprocessed, not into the original source code, in complicated cases it could be non-trivial to found the original cause in the metaprogram implementation. By utilizing the similarity of preprocessor metaprogramming and functional programming, we can use a purely functional language to ease the development and maintenance.

This paper is organized as follows. First, related results will be introduced in the next subsection. Then in section 2, the basic steps of preprocessor metaprogram modeling will be presented, along with detailed explanations in each subsection. Finally, section 3 concludes the paper.

Related work

The preprocessor subset of Boost library [6] provides a large number of utility macros. These include integer arithmetic operators, container data types, control structures like conditionals and loops. Metaprogramming in general is closely related to functional programming. Boost also includes a separate library for C++ template metaprogramming using functional constructs [1]. In [7], it is also shown that a functional language can be embedded into C++ by template metaprogramming.

2. Modeling preprocessor metaprograms

The properties of macros enable to implement them as functions in purely functional languages. This provides higher abstraction level by explicitly giving types to macro arguments and return values. Using data structures like tuples and lists also supports this. Testing and correction of errors also needs less effort this way. It is possible by the far more precise error reporting facilities of functional languages' compilers.

2.1. Method

The steps can be summarized as follows.

[1] For example, by redefinition using *#undef* and *#define*

[2] Such as avoiding recursive macro expansions

1. **Define the transformation.** How macros will be invoked by the user and what is the structure of the generated (expanded) source code.

2. **Create the functional model.** Use a restricted subset of a purely functional language. Test the functional implementation.

3. **Translate the model into macros.** Start with the macro-based representation of data types, then rewrite every other function as a macro.

2.2. Defining transformation

These steps will be presented on a simple program transformation, which extends the C language with a type class deriving mechanism similar to the one supported by the Haskell language [5]. This enables us to automatically generate code for specific behaviors of our custom data types, like equivalence checking or comparison. Haskell is also used to present the functional model of this example in later sections. The following example shows the usage of the envisioned macro system:

```
STRUCT( point
      , FIELD(int, x)
      , FIELD(int, y)
      , DERIVING(EQ, ORD, SHOW)
      )
```

which defines a custom data structure with two integer fields, and generates the code needed to support field-wise equality check, ordering comparison and conversion to string. The desired code after macro expansions is the following:

```
typedef struct { int x; int y; } point;

bool point_equals(point *a, point *b) { // EQ
    return int_equals(&a->x, &b->x)
        && int_equals(&a->y, &b->y);
}

int point_compare(point *a, point *b) { // ORD
    int r;
    if (0 != (r = int_compare(&a->x, &b->x)) return r;
    if (0 != (r = int_compare(&a->y, &b->y)) return r;
    return r;
}

void point_show(char *cstr, point *a) { // SHOW
    sprintf(cstr, "\t" "x" ": "); int_show(cstr, &a->x);
    sprintf(cstr, "\t" "y" ": "); int_show(cstr, &a->y);
}
```

The resulting code contains a standard C structure definition, and in this case, three additional function definitions. These are triggered by the arguments of *DERIVING* macro - each enumerated value corresponds to a specific function. The structure fields are also described with a special *FIELD* macro, because generated code refers structure field types and names independently. The variadic[3] *STRUCT* macro will generate all the resulting code, using data produced by the other macros. Note that generated code depends on the definitions of *T_equals*, *T_compare* and *T_show* functions for each field type *T*.

2.3. Functional model

We can model each macro with a function. For the sake of clarity, every function that represents a macro will be marked with a prime symbol.

```
struct' :: StructName → [Field] → Deriving → Code
struct' name fields deriving = mkStructDef ++ mkFunctions
    where ...
```

The first function takes a structure name, an arbitrary number of field descriptions and a deriving clause, and finally returns the generated code. It creates a structure definition based on its name and fields, and the corresponding function to each value in the deriving clause. While macros are generating tokens, this function basically composes a string. Typing is made to be more meaningful with type aliases:

```
type Code = String
type StructName = String
```

Since the two remaining macros, *FIELD* and *DERIVING* are representing data, their implementation will be detailed in the next section.

2.4. Translation of data types

Containers

The Boost library already supports four types of basic containers: tuples, lists, sequences and arrays [6]. These containers can be considered as polymorphic, because the macro system does not restrict the element types. It is possible to create heterogeneous lists and embed any container into another. Boost provides a large set of operations to build and manipulate these data structures, for example splitting a list into a head element and a tail list. For the sake of preprocessing performance, sequences are preferred over lists[4].

[3] A macro with variable number of arguments, available since C99[2]

[4] However, an empty list can not be represented with a sequence

Algebraic data types and constructors

Using previous containers we can simulate algebraic data types, which generally have the following form in Haskell [5]:

data T = $Cons_1\ Tp_{11}\ \ldots\ Tp_{1n}\ \mid\ \ldots\ \mid\ Cons_m\ Tp_{m1}\ \ldots\ Tp_{mk}$

where $Cons_x$ denotes the name of a constructor and Tp_{xy} is the type of one of its parameters. As every constructor can be represented as a function with the appropriate arity [3, 4], they can be implemented as a function-like macro with the same number of arguments.

#define T_$Cons_1(p_{11},\ldots,p_{1n})$ (T_$Cons_1$)$(p_{11})\ldots(p_{1n})$
...
#define T_$Cons_m(p_{m1},\ldots,p_{mk})$ (T_$Cons_1$)$(p_{m1})\ldots(p_{mk})$

Each constructor creates a sequence, where the first item is the name of the constructor function itself that can be used as a tag later. This makes pattern matching possible, which is explained in the next subsection. The rest of the sequence is simply holding the actual arguments provided to the call of the constructor. If constructor names are unique for all types, which is often a requirement in functional languages, the type name prefix of constructor macros can be omitted.

In our example, structure field declarations have a type and a name:

```
type FieldType = String
type FieldName = String
data Field = Field FieldType FieldName

field' :: FieldType → FieldName → Field
field' = Field
```

Again, type aliases are used to give more meaning to string types. Function corresponding to macro *FIELD* is a type constructor. It can be implemented according to the previous method as:

```
#define FIELD(type, name) (FIELD)(type)(name)
```

In our example, the different values for the deriving clause are forming an enumeration:

```
data ClassName = Eq' | Ord' | Show'
type Deriving = [ClassName]

deriving' :: [ClassName] -> Deriving
deriving' = id
```

As constructors of enumerations cannot hold additional data, they do not need to be represented with sequences:

```
#define EQ   EQ
#define ORD  ORD
```

```
#define SHOW SHOW
#define DERIVING(...) var_to_list(__VA_ARGS__)
```

Definition of *DERIVING* uses a special construction. It converts a comma separated variadic argument list to an ordinary list using *var_to_list*. This enables the users of the macro to specify an arbitrary number of parameters for it.

Typelessness of macros permits these translations to work even with recursive types, or types with higher kinds[5]. Translation of algebraic data types that are using record-like syntax could be done by creating a macro for each selector function.

2.5. Translation of control structures

Pattern matching

One of the main advantages of using algebraic data types is pattern matching. First, with this technique the execution paths of a single function can be separated based on the constructors of a parameter type. Second, constructor parameters can also be extracted in a single step:

f :: T → T'
f ($Cons_1\ p_{11}\ \ldots\ p_{1n}$) = ...
...
f ($Cons_m\ p_{m1}\ \ldots\ p_{mk}$) = ...

Separation of execution paths can be expressed with macros using the token pasting operator (##). We will need to fully expand its arguments before applying the token pasting operator. For this purpose, define *concat* as:

```
#define concat(x, y) concat_(x, y)
#define concat_(x, y) x ## y
```

then apply it to implement pattern matching scheme:

```
#define f(t) concat(f_, item_at(t, 0)) (to_args(tail(t)))
```
#define f_$Cons_1(p_{11},\ldots,p_{1n})$ = ...
...
#define f_$Cons_m(p_{m1},\ldots,p_{mk})$ = ...

where macro *tail* expands to the tail of a given sequence[6], and *to_args* converts a sequence to a comma-separated argument list.

Generally, a constructor function expands to a sequence that contains a tag as its first item, which is the same as the constructor's name. Since the C preprocessor does not support recursion in macros, reusing the name of a macro in its own definition will yield that name as an identifier token. This tag can be used together with the token pasting operator to form the name of an additional macro. The result of the pattern matching function will be given by this secondary macro that

[5] Types which have one or more type parameter
[6] A new sequence containing all elements of the original except the first

represents the execution path corresponding to the given constructor, and able to use its extracted parameters. Expansion of a macro with non-exhaustive patterns will likely give invalid results, as token pasting will generate a non-existing macro name.

Unlike in many functional programming languages, this pattern matching is not trivially compositional: further matching on a value of a constructor parameter (e.g. on p_{11}) needs to be defined with another separate macro. When matching is applied to more than one parameter, it also makes the system more complex.

In our example, pattern matching can be used to separate the code generation of different functions, based on values from the deriving clause.

Conditionals

A conditional expression always have a *then* and an *else* part in purely functional languages. It can be rewritten with pattern matching on condition, which can be translated to macros.

E = if *Condition* then $Result_{true}$ else $Result_{false}$ ⇒
E = F *Condition* where
    F $True$ = $Result_{true}$
    F $False$ = $Result_{false}$

Note that Boost library also has a set of conditional macros which are defined very similarly. *Case* expressions can also be rewritten into pattern matching likewise.

Iteration

While looping in its imperative meaning is unusual in stateless contexts, iteration of container types, like lists or sequences is needed. This is usually done through maps, filters and folds. Fortunately, the Boost library provides a large and very useful set of higher-order macros[7] to perform these tasks. In our example, fold can be used to iterate through field descriptors given to $STRUCT$ and generate code for the method bodies.

Let expressions and where clauses

With *let* expressions and *where* clauses we can create definitions that are local for a single expression or function:

```
let x = ... in ... x ...
f x = ... g ... where g y = ... x ...
```

If referential transparency holds for x, the first case is trivial and does not cause any problem. However, in the case of *where*, the local function g can use arguments of f, in this example x. As this kind of declaration nesting does not exist in the

[7] Macros which are parametrized with names of another macros

preprocessor language, macro g will also be global, and will have no access to x. To make translation to macros possible, rewrite the code above as the following:

```
f x = ... (g x) ...
g x' y = ... x' ...
```

3. Conclusion

Complex macro systems can be modeled by restricted functional programs. The higher abstraction level and better error reporting of functional compilers provides more rapid and safer development. Models implemented in functional languages can be translated back to macros almost effortless using the preprocessor subset of the Boost library, and the translation methods presented earlier. This translation leads to easily recognizable patterns in the metaprogram, which helps further maintenance. However, the developer who will maintain the functional implementation must know the functional language in which the model described.

As further work, a tool that can automate the translation process can be implemented. Creating macro libraries which are imitating the standard library of a concrete functional language is also a possibility, and they can further increase the readability of the translated code.

References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison Wesley Professional, 2004.

[2] ISO/IEC JTC1/SC22/WG14. *Programming Languages – C*. Standard. International Organization for Standardization, Dec. 1999.

[3] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. "Data types and pattern matching by function application". In: *Proceedings Implementation and Application of Functional Languages, 17th International Workshop, IFL05*. 2005.

[4] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. "Efficient Interpretation by Transforming Data Types and Patterns to Functions". In: *Trends in Functional Programming* 7 (2007), p. 73.

[5] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[6] *The Boost Library Preprocessor Subset for C/C++*. URL: `http://www.boost.org/doc/libs/1_55_0/libs/preprocessor/doc/index.html`.

[7] Porkoláb Zoltán and Sinkovics Ábel. "C++ Template Metaprogramming with Embedded Haskell". In: *Proc. 8th Int. Conf. Generative Prog. & Component Engineering (GPCE 2009)(New York, NY, USA)*. ACM, 2009.