C Preprocessor Macros from Functional Programs

Boldizsár Németh, Máté Karácsony, Zoltán Kelemen, and Máté Tejfel

Eötvös Loránd University, Dept. of Programming Languages and Compilers
H-1117 Pázmány Péter sétány 1/C, Budapest, Hungary
{nboldi,kmate,kelemzol,matej}@elte.hu

Abstract. The preprocessor of the C language provides a standard way to generate code at compile time. However, writing and understanding these macros is difficult. Lack of typing, statelessness, and uncommon syntax are the main reasons of this difficulty. Haskell is a high-level purely functional language with an expressive type system, algebraic data types, and many useful language extensions. These suggest that Haskell code can be written and maintained more easily than preprocessor macros. Functional languages have certain similarities to macro languages. By using these similarities this paper describes a transformation that translates lambda expressions into preprocessor macros. The existing compilers of functional languages are used to generate lambda expressions from the source code. As a result it is possible to write Haskell code that will be translated to preprocessor macros manipulating source code. This may result in faster development and maintenance of complex macro metaprograms.

Keywords: C preprocessor, functional programming, Haskell

1 Introduction

Preprocessor macros are fundamental language elements of C programming language. Using macros commonalities of abstractions can be expressed without runtime performance penalties. Developers can extend the capabilities of the host language without having to change the compiler itself. Possible usage are serialization, compile-time reflection or optimisation based on extra knowledge about the domain of the application or 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 modify any global, mutable information. When every macro has a single definition that does not change during preprocessing, 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 (e.g. avoiding recursive macro expansions). Although an expansion fails only when an inappropriate number of arguments are provided, the 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 find the error in the metaprogram implementation.

By utilizing the similarity of preprocessor metaprogramming and functional programming, a purely functional language can be used to ease the development and maintenance. Our solution automatically translates functional programs written in Haskell to C preprocessor macro definitions. It is implemented as a software tool named *hs2cpp*. This paper presents the ideas and applied practices, and the program transformation itself in a formalized way.

The rest of the paper is organized as follows. Section 2 describes the main architecture of our solution, and gives brief introduction to the source and target systems of the transformation. In Section 3 the transformation of basic data types and simple lambda expressions to macro definitions is formally defined. More advanced constructions, like recursion and higher-order functions are handled in Section 4. An example application of our method to compile time reflection is presented in Section 5. Our results including metrics of the compiled macro systems are presented in Section 6, while related metaprogramming tools, systems and practices are reviewed in Section 7. Finally Section 8 describes further ideas considered for later implementation, and concludes the paper.

2 Architecture

Our translation solution is designed as a plug-in in the Glasgow Haskell Compiler (GHC) [12]. Figure 1 summarizes the high-level architecture, including the most important data flows. The plug-in installs a Core-to-Core pass that does not change the program being compiled, rather creates C header files containing the generated macro definitions as a side effect. According to this design decision, the transformation is executed on the relatively simple Core syntax rather than the rich Haskell abstract syntax tree. The transformation (as mentioned earlier) takes advantage of the functional nature of the preprocessor macros. This is the reason why it is based on the Core representation instead of the C-- representation that is used by ordinary Haskell backends. For details about GHC's compilation pipeline see [7].

The next two subsections give a brief introduction to the source and target systems of the translation. As both of them are well-described areas, only those aspects are included which are the most important for understanding the translation logic. At the end of this chapter, subsection 2.3 presents the structure

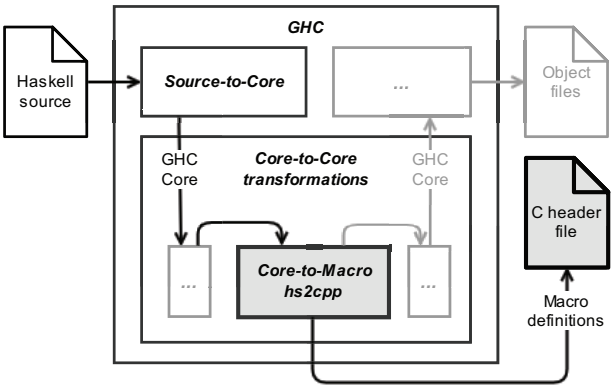

Fig. 1. High-level architecture

of the generated macro system, and introduces the notations used to define a transformation function.

2.1 GHC Core

In this section a simple λ-calculus is defined (see Table 1). The paper will use it in the description of transformations. This λ-calculus has a very schematic form without any special syntactic extensions. We are not assigning semantic rules or type system because the defined transformation is purely a syntactic transformation and easy to interpret for any specific lambda calculus.

Our solution generates preprocessor macros from GHC Core language that is based on an extended polymorphic lambda calculus. This calculus is named System F_C [18], and it is a practical compiler intermediate language based on System F [14].

Table 1. λ-calculus and GHC Core

	λ-calculus	GHC Core
variable	x	`Var x`
literal	l	`Lit l`
abstraction	$\lambda x.e$	`Lam x e`
application	$e_1 e_2$	`App` e_1 e_2
pattern matching	case e of $p_1 \rightarrow e_1$ $\vdots$ $p_n \rightarrow e_n$	`Case e v t` `[`m_1`,...,`m_n`]`
let expression	let $x = e_1$ in e_2	`Let (bnd x` e_1`)` e_2

`where` m_i `= (tag(`p_i`), var(`p_i`),` e_i`)`
`bnd n e = NonRec n e`

The representation of a program in GHC Core consists of many parts. For our translation the most important part is the list of bindings. Bindings contain executable code of the program. A binding can be simple (`NonRec`) or recursive (`Rec`). The simple binding contains a unique identifier and an expression and the recursive contains a list of name-expression pairs that can refer each other. The defined transformation manages only type-correct GHC Core representation which generated from a well-typed program.

There are six different constructions in GHC Core that are transformed:

- A *variable* contains a unique identifier that references an other definition.
- A *literal* can be a character, string, simple int, 64 bit int, float, etc. GHC Core distinguishes eleven different literals.
- An *abstraction* has a unique name bound to the argument of the abstraction.
- A *function application* contain two expression.
- *Pattern matching* is represented with the `Case` constructor. Patterns $\mathtt{m}_1, ..., \mathtt{m}_n$ are tuples with three elements: the tag of the constructor (`tag`), the captured variables (`var`) and the expressions that is evaluated when the pattern matches. There are three kinds of matches: ADT matches, literal matches and default matches. Core's pattern matchings are always complete. If the individual matches cannot cover the whole range of values, a default match will be generated. The `v` and `t` arguments are a binding for the pattern matched expression and it's type. They are generated for internal reasons, it will not affect the generated preprocessor code.
- A *let expression* `b` is a binding and `e` is an expression.

For technical reasons the transformation simplifies the GHC core representation, by removing local bindings and ignoring Cast and Tick GHC constructs as well as types and coercions.

Figure 2 introduces an example represented in Haskell, lambda calculus and GHC Core representation after some syntactic simplification. The example code has two parameters, that are natural (Nat) values. The return value is the sum of the two parameters.

2.2 The C preprocessor

As mentioned earlier, the C preprocessor operates by translating a token stream into another one. Although the preprocessor supports many directives, paper considers only file inclusion and macro definitions. Description of all supported directives could be found in the C99 standard [8], while [6] specifies the exact algebraic semantics of the preprocessor.

File inclusion is implemented through the `#include` directive, which allows to insert tokens from the specified file to the point where the directive occurs. With the `#define` directive, the programmer can assign a token sequence (called body) to a single-token name.

When the preprocessor detects a macro invocation, triggers an expansion: replaces the invocation with the tokens from its body. Paper will use "⇒" symbol

Haskell source code

```
add (Succ a) b = add a (Succ b)
add Zero     b = b
```

GHC Core

```
(Rec add (Lam ds (Lam b
          (Case (Var ds) wild typ
            [ ( (DataAlt Zero), [], (Var b) )
            , ( (DataAlt Succ), [a]
              , (App (App (Var add) (Var a))
                     (App (Var Succ) (Var b)))) ])))
```

λ-calculus

$$add = \lambda a.\lambda b.\,\mathrm{case}\,a\,\mathrm{of}\;\begin{array}{ll} Succ\ a & \rightarrow add\ a\ (Succ\ b) \\ Zero & \rightarrow b \end{array}$$

Fig. 2. Different representations of add

to notate macro expansions in the following code listings. Based on their arity, macros can be object-like (having no arguments) and function-like. In the second case, when a macro has parameters, their actual values will be substituted into its body upon expansion.

To guarantee its termination, the preprocessor keeps a list with the names of currently expanding macros. When the preprocessor detects a macro invocation in a list of tokens, it looks up if the expansion list contains the source of the tokens. When such situation is found, the macro invocation will not be expanded (see the example below).

```
#define REC(x) 1 + REC(x)
REC(2) ⇒ 1 + REC(2)
```

As a consequence, real recursion — including mutual — is not allowed so the preprocessor is not entirely Turing-complete. However, recursion can be simulated to a certain depth and branching can be done by the preprocessor. These suggest that the C preprocessor is able to emulate a calculation that is guaranteed to terminate in a given number of steps.

Although recursion is disabled, nested invocations of a macro is supported within its arguments. Before the expansion of a function-like macro, the arguments are scanned, and all possible expansions will be executed before their substitution. This process is called argument prescan. After the resulting tokens are substituted into the body, the scan happens once more, and possible macro invocations will be expanded too. Having this second scan, the argument prescan looks unnecessary. However, it has an important effect: during the prescan, the expansion list is not appended yet with the name of the currently expanding macro. Without this mechanism, the following nested substitutions would not happen:

```
#define ADD(x, y) ((x) + (y))
ADD(ADD(1, 2), 3)
 ⇒ ADD(((1) * (2)), 3)
 ⇒ ((((1) * (2))) + (3))
```

It is important to note that argument prescan makes preprocessing strict, as each argument will be expanded before it gets substituted.

There are two special operators in the preprocessor we need to mention. First, the stringification (`#`) operator creates a string literal from the tokens of a macro argument. It can be used for debugging and code generation purposes additionally to generating string literals. Second, token concatenation (`##`) could merge two individual tokens into a single one. For example, it makes possible to create macro names from multiple arguments, and expand them. Important to note in case of these operators, that argument prescan will not happen.

```
#define CONCAT_(x, y) x ## y
#define ONE 1
#define TWO 2
CONCAT_(ONE, TWO) ⇒ ONETWO
```

For this reason, sometimes a second expansion is needed to force the scan of their arguments:

```
#define CONCAT(x, y)  CONCAT_(x, y)
CONCAT(ONE, TWO)  ⇒ 12
```

As it will be detailed later in Section 3.7, this mechanism will have an important role in handling pattern matching expressions. It is the only way to branch in the body of a preprocessor macro.

2.3 The generated macro system

Haskell modules are compiled into C headers containing preprocessor macros. Import declarations between Haskell modules will be translated into preprocessor include directives. All the exported definitions of a Haskell module will became macros that are usable from the generated header file. A network of Haskell modules importing each other will became a network of header files. Like precompiled modules, pregenerated header files can be used by an included module. Existing macro definitions can be used in a new header file by including the generated headers. In Haskell code it is also possible to use macros defined earlier in a regular C header, thus invocation to existing macros can be integrated into a new, translated system.

Macro definitions in this paper are represented in their original syntax, with the addition of placeholders, surrounded by angle brackets. These templates can later be instantiated with different values to get concrete macro definitions. For example, the following template

```
#define ⟨name⟩(x) (⟨value⟩ + (x))
```

with substitution of `name = INCREMENT` and `value = 1` gives

```
#define INCREMENT(x) (1 + (x))
```

If the placeholder contains a translation expression, it must be replaced by the tokens that are the result of that expression. If the placeholder contains a name that is not substituted, than it is assumed to be a new unique name.

In the description of our transformations backslash (\) characters will be used to break overlong macro bodies into multiple lines, in accordance with the C preprocessor syntax.

3 Basic transformations

3.1 Primary Data and Operations

This paper reuses some ideas from the *Boost Preprocessing library* [1, **?**]. It provides a collection of various data structures and algorithms one can rely on: it supports representation and operations for tuples, sequences, boolean and integral values. For example, the next listing shows three numbers stored in different data structures.

```
#define TUPLE_OF_NUMBERS (1, 2, 3)
#define SEQUENCE_OF_NUMBERS (1)(2)(3)
```

A tuple simply stores a fixed number of items, which are accessed by their position in the tuple. Sequences are preprocessor-optimized lists, therefore most of their operations have linear complexity. Almost any kind of data could be stored in the place of numbers, including the possibility of nesting them into each other. As detailed later, these simple data types can be used to represent any kind of compound data, including abstract data types.

3.2 Objects in the Macro System

Three kinds of objects are represented in the system of generated macros. It can be decided if an object is a value, exception or thunk.

Values are data structures that represent a Haskell value. Values can be primitive types or algebraic data types. For example, the number 3 is represented with a sequence of two elements, one that identifies it as a value, and one that stores the actual number:

```
(VALUE)(3)
```

Thunks are functions that can be partially applied. A thunk stores the name of the macro that must be expanded when all arguments are present, the number of arguments needed and the arguments collected so far. For example, the function `(+3)` is represented as a sequence of four elements. The first element identifies it as a thunk, and the second specifies the macro that should be expanded when all arguments are present. Third number describes its arity, and the last is a sequence of arguments collected so far:

```
(THUNK)(PLUS)(2)(((VALUE)(3)))
```

Exceptions are errors that can be handled where a macro of the generated macro system is used. The representation of an exception is also a sequence. The first element identifies the object as an exception, and the second is the exception message:

```
(EXCEPTION)((Exception error message))
```

Exceptions appear in the generated code when the `error` function would be evaluated in the Haskell program. Common cases of exceptions include pattern matching errors, undefined behaviour (`undefined` or $\perp$) and illegal arguments given to a function.

Exceptions are manually propagated when an exception appears as a function inside a function application or when an exception is the subject of pattern matching. Exceptions are handled with the `IS_EXCEPTION` macro, which decides whether an object is an exception.

3.3 Representation of Haskell Values

To translate functional programs into preprocessor macros it is essential to find the correct substitutions of Haskell data structures. Fortunately Haskell data structures are built from simple constructs. The Transformation needs to represent primitive values and algebraic data types and encode these values respecting the lexical structure of the preprocessed file.

The Unit type, that is written as `()` in Haskell, has one possible value: Unit. It is simple to generate code from functions that receive a unit type. However, pattern matching on a unit type does not reveal any additional information, so it is rarely used.

```
#define ID(b) ID_ ## b
#define ID_UNIT UNIT
```

This is the simplest case of pattern matching, that will be described in more details in Section 3.7

Boolean values are also easy to implement. The range of boolean values is limited, therefore all operations can be implemented by a finite system of macros. For example, the logical negation can be implemented by concatenating the argument to a prefix, and generating two versions of the prefixed operation:

```
#define NOT(b) NOT_ ## b
#define NOT_TRUE FALSE
#define NOT_FALSE TRUE
```

Integral values can be represented as a number token. Integral operations can be done by concatenating these tokens to macro names. The same token pasting based approach can be used as for boolean values. Of course, this means that the range of integral values will be limited. For example, inspect the `INCREMENT` macro on the 2-bit representation of integral values. Incrementing the largest number does not change its value, to prevent errors when `INCREMENT` would be used multiple times on the maximal number.

```
#define INCREMENT(b) INCREMENT ## b
#define INCREMENT_0 1
#define INCREMENT_1 2
#define INCREMENT_2 3
#define INCREMENT_3 3
```

Characters can be represented by a token containing a single character. However it is not possible to use lists of characters to generate text, as it can be seen in Section 3.4.

Floating point values cannot be represented properly because of the large number of these values, and the lack of precise semantics of the operations.

Algebraic Data Types (ADTs)[13] are the building blocks of all complex types in Haskell. In fact, the Unit and the Boolean types shown above are examples of ADTs. Values of algebraic data types can be created using the constructors of the data type. Each constructor has a number of argument types. A value with an ADT can be represented as a tag and a list of values. The tag identifies the constructor that is used to create the value. The values are the arguments of this constructor. For example the representation of the ADT value `Just 3` is the following:

```
(VALUE)((JUST_TAG ,((VALUE)(3))))
```

In fact, Haskell types that are normally taken as primitive (`Int`, `Char`) are also ADTs

3.4 Storing textual information

One of the goals of our system is to provide a high-level abstraction for code generation. Strings are lists of characters in Haskell. There are two problems with storing textual information in Strings.

First, our macro system operates on the level of tokens. It means that white space cannot be represented in our model. Second, storing a list where all element is a single character comes with a big overhead.

An alternative type for creating large sections of text is provided, for example function definitions generated into the source code. Because of the large amount of data that is stored in values of this type, the transformation uses a representation with less overhead than a list of characters that would be the direct translation of a Haskell String.

The name of this data type is *TokenList*. The construction of TokenLists is limited in Haskell. Text literals can have TokenList type. Two new operators are introduced in Haskell, that resemble operators of the preprocessor. Two TokenLists can be concatenated by the `#` operator. Two tokens can be used to create a new token by the `##` operator. These two tokens must for a valid token together, for example `"f" ## "()"` is not a valid expression.

```
"a b" # "c d" ⇒ "a b c d"
"a" ## "b" ⇒ "ab"
```

It is important to state that a Haskell String cannot be converted into a TokenList, because it would lose it's white space information. It would be impossible to tell if two characters belong to the same token in the representation of a Haskell String.

There are some limitations about the stored textual information. These limitations are necessary to respect the lexical rules of the generated file.

Every value must contain valid parentheses. For example the literal `"("` is not allowed, because it contains an unbalanced parenthesis. The same restriction is true for quotes. There are situations when the generated code must be placed between parentheses, for example to generate a parameter list. Generated code between quotes is also needed, for example, to generate string literals in code. The `paren` and `quote` Haskell helper functions can solve these situations. In the generated code they are replaced by macros that can do the transformation directly.

Additionally, the `#` and `##` preprocessor operators cannot be part of the generated code, but this is not a problem in practice, because they cannot be the part of a language that is preprocessed.

3.5 Strictness of the generated macros

As it is known, the evaluation of a Haskell program is lazy by default. It means that an expression is only evaluated when the value of the expression is used. This means that exceptions and non-terminating subexpressions can be found in a terminating expression if they are not necessary to be evaluated.

As it was mentioned in Section 2.2, the evaluation of macros is strict, the arguments are expanded first, and then they are inserted into the macro body.

The evaluation of the generated macro system will be strict, but in practice it allows exceptions to appear in expressions if they are not evaluated. Because the preprocessor always terminates, the generated macro system will not contain infinite recursion or infinite data structures. Non-evaluable objects can only be created by explicit `error` call. The `error` call creates an exception object, that can be passed between functions without affecting control flow.

Even if the underlying preprocessor uses strict evaluation, our generated macro system behaves in a way that is similar to lazy evaluation.

3.6 Eliminated and ignored constructs

Local bindings like let $x = expr_1$ in $expr_2$ are eliminated before the transformation begins. They are replaced with top-level bindings, and any implicitly passed data becomes explicitly given.

```
f p = let a = p          ⇒    f p = (a p) + 12
      in a + 12               a p' = p'
```

A Cast expression performs a type coercion on the received expression. Because the Haskell type system is not used during the transformation, Cast ex-

pressions are simply ignored and only the expression that is affected by Cast is transformed.

Tick expressions are used to enable debugging and profiling operations on the compiled Haskell program. Because the transformation does not provide debugging and profiling support, Tick expressions are also ignored and only the expression that is inspected by the Tick is transformed.

Because GHC Core representation contains type application, it represents types as expressions with the Type constructor. It can also represent type Coercions with the Coercion constructor. Because such constructs are not needed, they are completely ignored while generating code.

3.7 Mapping of λ-expressions to Macros

The representation of an expression is a list of tokens that will be placed where the expression is used. Compiling the expression can also generate macro definitions, that will be inserted into the generated file sequentially. The transformation can be formalized as a function where *expr* is a lambda expression and *macro* is a list of C tokens, and $macrodef$ is a list of C preprocessor directives:

$$\varphi : expr \rightarrow (macro, macrodef)$$

It is practical to also define the transformation on bindings. A single binding (*bnd*) can be transformed into a system of macros:

$$\varphi' : bnd \rightarrow macrodef$$

A variable will be transformed into a token containing the unique name of the variable. The unique name prevents multiple variables or definitions with the same name interfering with each other in the generated macro system.

Literals will be transformed to the representation of their value.

Function application will be performed by adding a parameter to the thunk of the applied function. After application if the thunk is satisfied (it received the required number of arguments) the macro stored in the thunk will be called. Otherwise the argument is added to arguments collected in the thunk.

$$\varphi(fe) = \texttt{APPLY(}\langle\varphi(f)\rangle\texttt{,}\langle\varphi(e)\rangle\texttt{)}$$

`APPLY` function is a helper function that adds a new argument to a (partly applied) function. If the function is satisfied with the new element, it performs the application by calling the macro that is stored in the thunk. Otherwise it just stores the new argument in the collected arguments of the thunk.

A lambda abstraction is transformed into a thunk that calls a macro when the argument is applied to it.

$$\begin{aligned}\varphi(\lambda x.e) = {} & \texttt{(THUNK)(}\langle body\_id\rangle\texttt{)(1)()} \\ & \texttt{\#define } \langle body\_id\rangle\texttt{(x) } \langle\varphi(e)\rangle\end{aligned}$$

The representation of pattern matches can be divided into three steps. First it must be checked that the object matched on is a value. Second, if it is an exception, it must be propagated unchanged. Finally, the tag must be concatenated to a unique prefix for branching on different cases. If the matched value has algebraic data type, the tag is explicitly present. If it is a primitive value, the value is used as a tag. The macros generated for cases are based on the expressions for each case.

$$\varphi\left(\begin{array}{l}\text{case } e \text{ of } p_1 \to e_1 \\ \qquad\qquad \vdots \\ \qquad\qquad p_n \to e_n \\ \qquad\qquad \_ \to e_{def}\end{array}\right) = \langle try\rangle(\langle\varphi(e)\rangle)$$

```
#define ⟨try⟩(⟨x⟩) IF(IS_EXCEPTION(⟨x⟩), x, ⟨match⟩(x))
#define ⟨match⟩(⟨x⟩) IF(IS_COVERED(TAG(⟨x⟩), ⟨tag(p1)...tag(pn)⟩), \
                        ⟨dispatch⟩(x), ⟨default⟩)
#define ⟨dispatch⟩(⟨x⟩) ⟨prefix⟩_ ## TAG(⟨x⟩)ARGS(⟨x⟩)
#define ⟨prefix⟩_⟨tag(p1)⟩(⟨args(p1)⟩) ⟨φ(e1)⟩
...
#define ⟨prefix⟩_⟨tag(pn)⟩(⟨args(pn)⟩) ⟨φ(en)⟩
#define ⟨default⟩ ⟨φ(edef)⟩
```

The `IS_COVERED` helper function decides whether the tag given during macro expansion is explicitly handled by this pattern match or becomes a default case. The `TAG` function extracts the tag from the representation of an ADT while the `ARGS` extracts its arguments.

3.8 Data types and constructors

In Haskell data constructors are used to create values of algebraic data types. The translator constructs thunks from calls of constructors that have arguments, and values from the ones that does not have. After enough parameters are given the ADT representation will be constructed as seen in Section 3.3. The next listing shows code generated from data constructors `Bool` and `Pair`. Even when the constructor has no arguments the representation contains a comma to separate the tag and the (empty) sequence of arguments.

```
data Bool = True | False
data Pair a b = Pair a b

#define True (VALUE)((True_TAG,))
#define False (VALUE)((False_TAG,))
#define Pair_CTOR(a1,a2) (VALUE)((Pair_TAG,(a1)(a2)))
#define Pair (THUNK)(Pair_CTOR)(2)()
```

4 Advanced constructions

4.1 Keeping scope

One problem that appears when the bindings are transformed into macro definitions is that the scope of the bindings are changed by the transformation. For example inspect the transformation of the $const = \lambda ab.a$ function.

```
#define const      (THUNK)(const1)(1)()
#define const1(a)  (THUNK)(const2)(1)()
#define const2(b)  a
```

The problem is that a is not in scope when the `const2` macro is expanded. This problem is solved by manually passing arguments that are in scope in the Haskell program but are out of scope in the generated macro definition. This happens in the case of abstractions and pattern matches. The example previous will be transformed like the following:

```
#define const        (THUNK)(const1)(1)()
#define const1(a)    (THUNK)(const2)(2)((a))
#define const2(a, b) a
```

4.2 Problem with recursive macros

In these sections the nested expansion of macros will be called as a *call chain*. A *macro family* is the collection of macro definitions generated to evaluate a given function.

There are cases when a macro call would be placed inside the body of the same macro. As Section 2.2 shows, the inner macro will not be expanded. This will be even a problem with the helper functions in our system. For example the `APPLY` function will be called in the outermost expression and in the definition of `APP_BODY` when the following function will be generated.

```
let app f a = f a in app id x

#define APP_BODY(f,a) APPLY(f,a)
#define APP = (THUNK)(APP_BODY)(2)()
```

Lets look at how the `APP` macro is expanded.

```
APPLY(APPLY(APP , ID) , ⟨x⟩)
 ⇒ APPLY((THUNK)(APP_BODY)(2)((⟨ID⟩)) , ⟨x⟩)
 ⇒ APP_BODY(⟨ID⟩ , ⟨x⟩)
 ⇒ APPLY(⟨ID⟩ , ⟨x⟩)
```

Here the `APPLY` macro should be expanded from a token that was produced by the expansion of the same `APPLY` macro. If multiple definitions of the macro are created, than $\texttt{APPLY}_1$ could be used in the first step and $\texttt{APPLY}_2$ in the third.

The transformation automatically detects that the two invocations must use a different version of the `APPLY` macro. It generates at least as many independent definitions for the macro as needed.

4.3 Recursion

The representation of recursive functions are macros that are expanded into a token list that can contain the invocation of the same macro.

$$\varphi' \left(\begin{array}{l} map = \lambda f.\lambda ls.\, \mathrm{case}\, ls\, \mathrm{of} \\ \qquad\qquad\qquad\qquad \mathrm{nil} \rightarrow \mathrm{nil} \\ \qquad\qquad\quad \mathrm{cons}\ c\ r \rightarrow \mathrm{cons}\ (f\ c)\ (map\ f\ r) \end{array} \right) =$$

```
#define CASE_nil() nil
#define CASE_cons(f,c,r) \
        APPLY(cons,APPLY(f,c),APPLY(map,f,r))
#define DISPATCH(ls,f) CASE_ ## TAG(ls) APPEND(f, ARGS(ls))
#define map_BODY(f,ls) TRY(DISPATCH,ls,f)
#define map (THUNK)(map_BODY)(2)()
```

Here the name of the `map` macro appears in the body of the `CASE_cons` macro. The invocation of the `CASE_cons` macro is expanded from the `map` macro itself.

The solution to the problem is the same as in the case of the `APPLY` macro before. The only difference is that in this case the author of the Haskell function have to estimate how many of times his function can appear in a call chain. There is a default number, but it can be overridden by the author. In the case of the map function, this is simple, as it is maximal length of a list that can be mapped. When the generated macro would be used in a way that exceeds the maximum number of nested applications, an exception will be raised.

4.4 Higher-order functions

One problem with higher-order functions is that they can receive themselves as arguments. For example take the following expression `map (map f) ls`. In the generated code the outer map function will receive the thunk of the partially applied map function `map f`. When the inner map is applied the first time the macro expansion will not happen because the application of the outer map already used the `map` macro.

Different macros (families of macros) can be generated for every use of the higher-order functions. For mentioned example, `map (map f) ls` would be translated into:

```
APPLY(map¹, APPLY (map², f), ls)
```

However, there is a better solution: first analyse the representation and inspect if one instance of the higher-order function can call the other instance. Different macros must only be generated when one function can call the other. This modification reduces the number of macros generated, but the evaluation will behave in the same way. This method generates false positives and can only be used inside a compilation unit.

The final solution to this problem is to track function use dynamically. Information about function usage can be passed as an argument between calls, and

it can be checked what is the first macro definition that is not used already in the call chain. This will cause a large overhead when evaluating macros of the generated macro system.

There is another problem when using higher-order functions. If they receive another function as an argument it can be evaluated multiple times, even if it is not a recursive function. For example, take the `until` function that applies the given `f` function until a `p` condition is satisfied.

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f = go
  where go x | p x       = x
             | otherwise = go (f x)
```

These problems are similar to the problem with higher order functions receiving themselves as arguments, and the possible solutions are the same. If the `f` function is not recursive, but used in a higher order function the user must be able to specify how many times can it be evaluated.

4.5 Annotating definitions

There are cases when the transformation needs user-defined information to generate macros. For example such information is the maximal call depth of recursive functions. Two ways were inspected to give such information to the macro generator plug-in.

By using annotation pragmas it is possible to store any value as an annotation for a top-level binding, type constructor or module. For example, annotate the recursive function `add` to be able to run for 10 recursive calls. It looks like the following in Haskell code:

```
{-# ANN add (Recursive 10) #-}
add :: N -> N -> N
add Z n = n
add (S n) m = add n (S m)
```

In some cases this approach does not work properly. Core-to-Core transformations can rewrite expressions and create new bindings. For example, in some situations GHC collects mutually recursive functions into a tuple. The original definition only selects a function from this tuple. In this case there is no guarantee that the annotation will be found on the definition that was annotated in the source code.

Another solution to annotate definitions uses type classes to provide information for transforming the given definition. This information will be encoded in a type constraint that is always satisfied. This type class of the constraint does not serve any other purpose than to label a definition with arbitrary information.

```
class Recursive (i :: Nat)
instance Recursive i

add :: Recursive 100 => N -> N -> N
```

```
add Z n = n
add (S n) m = add n (S m)
```

When the add function is transformed, its type can be queried. The context of the type is analysed and constraints meaningful for the translation are collected. The result will be that 100 instances will be generated from the add function.

5 Example: compile time reflection

This section shows how the transformation can be used to generate macros that automatically derive functions for any user-defined C data structure, using its field definitions. These structural functions can include member-wise comparison of structure instances for equality or ordering, copy and assignment, or converting the object to its textual representation. The macro system generated from the following Haskell definitions emulates the Haskell-like deriving behaviour for C programs at compile time. In the example only the important definitions are introduced.

The following C code will define a structure named `point` with two integer fields, called `x` and `y`. It also derives a function which is able to convert a `point` to a regular C string. The second listing shows the expected code after all macros expanded.

```
mkStruct( point
        , Field(int, x), Field(int, y)
        , Deriving(ClassShow) )
```

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

As C does not support compile time reflection, this can only be implemented by supplying all information about the structure as arguments for the macro that generates the code. This is why `mkStruct` receives the name of the structure and information about an arbitrary number of fields. Type and name information is encapsulated by a `Field` macro invocation for each member. The last argument of `mkStruct` expresses the request to derive a member-wise string conversion function for this data structure, using `Deriving` and `ClassShow` macros. These last three macro names are starting with upper case letters, in fact, they will be data constructors in the Haskell code as expected by their name. Only `mkStruct` will generate code using the data provided.

While the generated macro system will generate C code, the return value is typed as a TokenList. The following type aliases are defined for convenience.

```
type Code = TokenList
type StructName = TokenList
```

Field data and deriving clauses are represented with algebraic data types. The `FieldType` and `ClassName` types are enumerations.

```
data FieldType = FTInt
type FieldName = TokenList
data Field = Field FieldType FieldName

data ClassName = ClassShow
data Deriving = Deriving [ClassName]
```

The code generation is divided into two steps. The `#` operator is used to concatenate the code for the structure definition and the derived string-conversion function.

```
mkStruct :: StructName -> [Field] -> Deriving -> Code
mkStruct name fields (Deriving classes)
  = mkStructDef # mkFunctions
```

The structure definition is simply assembled from the structure name and from its field definitions. Functions defined below this point are in the `where` clause of `mkStruct` to implicitly receive its arguments.

```
mkStructDef :: Code
mkStructDef = "typedef struct {"
            # mkFieldDefs # "}" # name # ";"

mkFieldDefs :: Code
mkFieldDefs = foldl singleFieldDef "" fields

singleFieldDef :: Code -> Field -> Code
singleFieldDef code (Field fType fName)
  = code # getTypeName fType # fName # ";"

mkFunctions :: Code
mkFunctions = concat (map singleFunction classes)
```

The `singleFunction` calls the appropriate generator function for the selected class. As there is only one possible class name in this example, it is fairly simple. Code sections that must be put inside parentheses must be given to the `paren` function (see Section 3.4 for explanation).

```
singleFunction :: ClassName -> Code
singleFunction ClassShow = mkShowFunction

mkShowFunction :: Code
mkShowFunction = "void "
  # name ## "_show" # paren (name # " *a")
  # "{" # (foldl mkShowStmt "" fields) # "}"
```

String literals can be generated by using the `quote` function. Similarly to parentheses, quotes cannot appear in string literals.

```
mkShowStmt :: Code -> Field -> Code
```

```
mkShowStmt code (Field fType fName) = code
  # "printf" # paren (quote (fName # ":"
  # getFmtString fType# ", a->" # fName) # ";"

getFmtString :: FieldType -> MacroToken
getFmtString FTInt = "%d"

getTypeName :: FieldType -> MacroToken
getTypeName FTInt = "int"
```

For the recursive helper functions foldl, map and concat it is important to specify how deep the recursion will be. In this case, it shows how long lists can be manipulated by these functions. For this example, this number will limit how many fields, and values in deriving clauses can be used.

6 Results

The size of the generated macro system was inspected. The number of generated macros depend on the built-in data types, constructors and bindings.

For the number of macros generated from bindings there are several things need to be considered. Basically the number of macros generated depends on the complexity of the bound expression. In case of recursive bindings the user can configure the maximal depth of the recursion. This will multiply the number of macros generated for the binding, because macro definitions need to be independent in every level of call depth.

The translation was tried on 40 functions, collected from the list-related functions of Haskell Prelude that are described in [11]. The definition of these functions was written in 140 effective lines of code. The maximal recursion depth 10 was used for these functions. As a result there are approximately 2200 lines of macro definitions generated from these functions, 30000 lines of helper functions and 403 lines of macros generated from the built-in data constructors.

7 Related work

As mentioned earlier, the *Boost Preprocessing library* [1] provides macros for handling various data structures, including tuples, arrays, lists, sequences. Operations on boolean and integral values, like negation, addition and subtraction in limited ranges is also supported. It also provides emulation of conditional and loop control structures. Some functional operations, like filter, map and fold is also supported. This also shows that functional elements can contribute to writing preprocessor macros. Despite its rich features, it still could be hard to develop and debug complex preprocessor metaprograms using this library.

For platforms where not only C, but C++ compilers are also available, there are further options. Earlier researches are showed that C++ templates are Turing-complete [19]. In [15], Zoltán Porkoláb describes the connection between functional programs and C++ templates. There is a chapter about embedding

Haskell into C++ [16], and the conversion of these embedded functional programs using the internal representation of YHC, the York Haskell Compiler [4]. There are certain similarities between his method and our approach. First the YHC compiler is used to translate Haskell code sections, embedded into C++ code, into Yhc.Core representation. Then it is translated to a language called *Lambda*, defined by the author. *Lambda* code is finally translated into template metaprograms.

The *Boost library* also includes a template metaprogramming library, called *MPL* [5]. It supports programming with similar constructs and data types as the preprocessor library shown earlier. The interface of this collection resembles the C++ Standard Template Library. Using template metaprograms as the generated language solves many problems our implementation must handle, for example, template metaprograms are enabled to be recursive. However, it is limited to C++ and cannot generate arbitrary code.

Starting from the C++ 11 standard [9], the *constexpr* specifier allows compile-time usage of variables and functions [17]. The specification supports limiting the call depth of *constexpr* functions. In case of the solution presented in this paper, it was a necessity. The wide-spread GCC compiler supports compile-time programming with memoization. However, full constexpr support is available only from GCC version 4.7. This prevents legacy systems with older compilers from taking advantage of this method of compile-time programming.

Another approach to support metaprogramming in C is to extend the language itself. Meta-C [10] introduces new programming elements into the language while it remains fully backwards compatible with the C99 standard [8] its based on. It provides Turing-complete tools for analyzing and manipulating C programs at compile-time. The language was implemented by its own, custom compiler. According to the project's website [3], the last release was an alpha version in 2007.

Other macro languages have semantics that are different. For example the M4 preprocessor [2] is more expressive than the C preprocessor. It allows recursion which was a serious problem when specifying the transformations. It also allows the creation of new definitions while evaluating macros.

8 Conclusions

This paper presents a method to automatically translate Haskell programs into C preprocessor directives. There is no dependency on compiler or library versions, so the generated macros can be used in legacy projects to replace hand-written macros or other preprocessing tools. The solution is implemented as a plug-in that integrates into the Glasgow Haskell Compiler. Because the simple Haskell Core representation is used for the transformation, all language extensions can be used to generate preprocessor macros.

The basic mapping between the Core representation and preprocessor macros is described. However this simple mapping cannot handle the differences between Haskell and preprocessor scoping rules, nested function application, recursive and

higher-order functions. This paper presents solutions for each of these problems, although for recursion and higher-order functions these solutions are limited.

Our implementation applies a specific language and compiler, but the defined transformation is generic. It can be used on any programming language that can be simplified to a λ-calculus.

Solutions for special cases of higher-order and recursive functions were provided in the Section 4, but the general problem is yet to be solved. Some approaches were provided to this issue, but generating reliable code requires further research.

References

1. Boost Preprocessor library. `http://www.boost.org/doc/libs/1_57_0/libs/preprocessor/doc/index.html`, accessed: 2015-02-20
2. M4 preprocessor language. `http://wolfram.schneider.org/bsd/7thEdManVol2/m4/m4.pdf`, accessed: 2015-02-20
3. The MetaC Language. `http://www.maier-komor.de/metac/`, accessed: 2015-02-20
4. York Haskell Compiler. `https://wiki.haskell.org/Yhc`, accessed: 2015-02-20
5. Abrahams, D., Gurtovoy, A.: C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond. Pearson Education (2004)
6. Garrido, A., Meseguer, J., Johnson, R.: Algebraic semantics of the C preprocessor and correctness of its refactorings (2006)
7. Jones, S.L.P., Hall, C., Hammond, K., Cordy, J., Kevin, H., Partain, W., Wadler, P.: The glasgow haskell compiler: a technical overview (1992)
8. JTC1/SC22/WG14, I.: Programming Languages – C. Standard, International Organization for Standardization (12 1999)
9. JTC1/SC22/WG21, I.: Programming Languages – C++. Standard, International Organization for Standardization (09 2011)
10. Maier-Komor, T., Färber, G.: Metac: A metaprogramming extension to c enabling crosscutting reconfiguration of embedded software
11. Marlow, S.: Haskell 2010 language report
12. Marlow, S., Jones, S.P., et al.: The glasgow haskell compiler (2004)
13. Paul Hudak, John Hughes, S.P.J.P.W.: A history of haskell: Being lazy with class. In: Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III). pp. 14–15 (2007)
14. Pierce, B.C.: In: Types and Programming Languages. pp. 341–344. MIT (2002)
15. Porkoláb, Z.: Functional programming with c++ template metaprograms. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) Central European Functional Programming School, Lecture Notes in Computer Science, vol. 6299, pp. 306–353. Springer Berlin Heidelberg (2010), `http://dx.doi.org/10.1007/978-3-642-17685-2_9`
16. Porkoláb, Z., Sinkovics, Á.: C++ template metaprogramming with embedded Haskell. In: Proc. 8th Int. Conf. Generative Prog. & Component Engineering (GPCE 2009)(New York, NY, USA), ACM. pp. 99–108 (2009)
17. Stroustrup, B.: Programming: principles and practice using C++. Pearson Education (2014)
18. Sulzmann, M., Chakravarty, M.M., Jones, S.P., Donnelly, K.: System F with type equality coercions. In: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation. pp. 53–66. ACM (2007)
19. Veldhuizen, T.L.: C++ templates are turing complete. Available at citeseer. ist. psu. edu/581150. html (2003)