Computing and Informatics, Vol. 35, 2016, 819–851

View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Computing and Informatics (E-Journal - Institute of Informatics,...

DEFINING C PREPROCESSOR MACRO LIBRARIES WITH FUNCTIONAL PROGRAMS

Boldizsár Németh, Máté Karácsony
Zoltán Kelemen, Máté Tejfel

Eötvös Loránd University
Department of Programming Languages and Compilers
H-1117 Pázmány Péter sétány 1/C, Budapest, Hungary
e-mail: {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 expressive type system, algebraic data types and many useful language extensions. These suggest that Haskell code can be written and maintained easier 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. Existing compilers for functional languages generate lambda expressions from the source code as an intermediate representation. As a result it is possible to write Haskell code that will be translated into preprocessor macros that manipulate source code. This may result in faster development and maintenance of complex macro metaprograms.

Keywords: C preprocessor, functional programming, Haskell, program transformation, transcompiler

Mathematics Subject Classification 2010: 68-N15

1 INTRODUCTION

Preprocessor macros are fundamental elements of the C programming language. Besides being useful to define constants and common code snippets, they also allow

developers to extend the capabilities of the language without having to change the compiler itself. Possible usage are serialization, compile-time reflection, 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 generally does not allow to express programs in a straightforward way or using a higher level of abstraction. These problems 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 lists of tokens 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 most cases it is 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 by modeling macro definitions with functional programs [1]. Our solution is to automatically translate 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 techniques and the program transformation in a formalized way.

While we considered writing a small DSL to generate macros from, we decided to use Haskell instead. As it can be seen later, some features of Haskell can only be used with certain limitations when generating preprocessor code. Nevertheless, we found strong arguments for using Haskell with GHC as a compiler. For example, existing compiler features (parser, type checker, etc.) can be reused. Haskell is relatively well-known and has a plug-in system that lets compilation steps to be registered, working on the Core representation. This enables the use of Haskell's rich set of language features and extensions without having to define translation rules for them, since they are transformed into simpler constructs by the time the preprocessor code is being generated.

The rest of the paper is organized as follows. The next subsection shows the motivation behind writing complex macro libraries through an example. It also reviews how the proposed workflow of *hs2cpp* can ease the development of such libraries. 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 is presented in Section 5. Related metaprogramming tools, systems and practices are reviewed in Section 7. Section 8 describes further ideas considered for later implementation, and concludes the paper. Finally, Section 9 presents the complete transformation of simple calculations from the Haskell program to the final result.

1.1 Motivation: Program Transformation Using Macro Systems

The common usage of the C preprocessor is to define constants and simple parametrized expressions, which are expanded by substitution during compilation depending on the actual options. However, the preprocessor could also be used to implement complex source code transformations. To understand how to achieve this with the preprocessor, take a look at how conventional source-to-source transformations work.

Source-to-source transformations are usually described with a set of transformation rules. A transformation is executed by the application of these rules on an input source code according to a strategy. A rule usually contains a pattern with variable definitions and a replacement that uses the variables defined in the pattern. When the pattern matches a fragment of the input source code, the variables are assigned with the corresponding elements. Then the matched fragment will be replaced with the substituted replacement defined by the rule. This process could be split up into the following steps:

1. Find code fragments matching the searched pattern.
2. Assign values for pattern variables.
3. Substitute pattern variables into replacement.
4. Replace the matching code fragment with the new replacement.

It is easy to see that the last two steps could be executed by a preprocessor using function-like macros. The first two steps are done by the programmer, who uses these function-like macros in the appropriate places in code instead of writing C declarations, statements and expressions directly. When carefully designed, the macro system corresponding to a transformation rule set could be used as domain specific language embedded into C or serve as language extension to C. For example, it is possible to create a macro system that extends C with a deriving mechanism similar to Haskell's:

```
STRUCT(point,
       FIELD(int, x),
       FIELD(int, y),
       DERIVING(CLASS_EQ, CLASS_ORD, CLASS_SHOW)
       )
```

Expansion of the macro invocations above could result in a regular C data structure definition and three function definitions that implements equality, ordering and pretty printing for the data structure. The structure of these macro invocations implies that all macros except `STRUCT` resemble constructors of algebraic data types, that could be implemented easily in Haskell:

```
type FieldType = String
type FieldName = String
data Field = Field FieldType FieldName
data ClassName = ClassEq | ClassOrd | ClassShow
data Deriving = Deriving [ClassName]
```

Using these types, the definition of the macro `STRUCT` could be given as a function definition named `struct` in Haskell:

```
type Code = String
type StructName = String
struct :: StructName -> [Field] -> Deriving -> Code
```

Implementation of these macros can be very challenging, as the representation and processing of algebraic data types is not trivial using the preprocessor. To ease the development of similar macro systems and libraries, *hs2cpp* proposes the workflow shown in Figure 1.

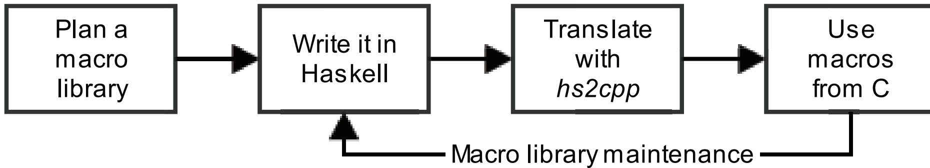

Figure 1. Workflow of macro library development with hs2cpp

hs2cpp is primarily intended to be used to prototype program transformations which could be executed by a C preprocessor. To ease the development of complex macro systems, it allows the user to write the transformations in Haskell, then translate them into macro libraries. These libraries then can be used by C programmers easily.

Section 5 details a more complex example, while Section 9 contains some simple calculation written in Haskell, along with their corresponding macro systems generated by *hs2cpp*, including examples of invocation and expansion of these macros.

2 ARCHITECTURE

Our translation solution is designed as a plug-in for the Glasgow Haskell Compiler (GHC) [2]. Figure 2 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 [3].

As the implementation of the translation is a GHC plugin it can be used by invoking GHC with a special flag to indicate the used plugin. The translation can be integrated into any higher-level build solution.

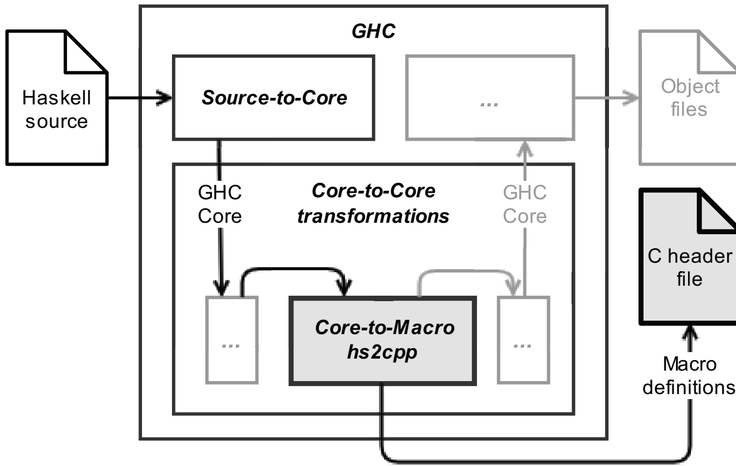

Figure 2. High-level architecture

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 that are the most important for understanding the translation logic. At the end of this chapter, Subsection 2.3 presents the structure of the generated macro system, and introduces the notations used to define a transformation function.

2.1 GHC Core

In this section a simple extension for the λ-calculus is defined (see Table 1). The paper will use this formalism in the description of transformations. This extended λ-calculus has a very schematic form without any special syntactic extensions. We are not using 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 [4], and it is a practical compiler intermediate language based on System F [5].

	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 e1 e2`
pattern matching	$\text{case}\, e \,\text{of}\; p_1 \to e_1$ $\vdots$ $p_n \to e_n$	`Case e v t` `[m1,..., mn]`
let expression	$\text{let}\, x = e_1 \,\text{in}\, e_2$	`Let (bnd x e1) e2`
non-recursive binding	–	`NonRec n e`
recursive binding	–	`Rec bnds`

```
where mi = (tag(pi), var(pi), ei)
      bnd n e = NonRec n e
      tag(pi) = DataAlt ctor
              | LitAlt lit
              | DEFAULT
```

Table 1. λ-calculus and GHC Core

The representation of a program in GHC Core consists of multiple parts. For our translation the most important part is the list of bindings. Bindings contain evaluable expressions for the execution of the program. A simple binding contains a unique identifier and an expression. A recursive binding group contains a list of name-expression pairs that can refer each other. The defined transformation only handles type-correct GHC Core representation that is created from a well-typed program.

There are six different constructions in GHC Core that are transformed. These constructions and the corresponding λ-expressions can also be seen in Table 1.

- A *variable* contains a unique identifier that references a definition.
- A *literal* can be a character, string, simple int, 64 bit int, float, etc. GHC Core distinguishes eleven different kinds of literals.
- An *abstraction* has a unique name bound to the argument of the abstraction.
- A *function application* contains two expressions.
- *Pattern matching* is represented by the `Case` constructor. Patterns $m_1, \ldots, m_n$ are tuples with three elements: the tag of the constructor (`tag`), the captured variables (`var`) and the expression that is evaluated when the pattern matches. There are three kinds of matches: algebraic data type 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 during desugaring. The `v` and `t` arguments contain GHC-specific information and they will be ignored by the transformation.

- A *let expression* is composed of a binding `b` and an expression `e`.

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

$$add = \lambda a.\lambda b.\,\mathrm{case}\, a \,\mathrm{of}\ \begin{array}{l} Succ\ a \to add\ a\ (Succ\ b) \\ Zero \to b \end{array}$$

Figure 3. Different representations of add

Figure 3 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 values created by constructors `Succ` and `Zero`. 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, this paper considers only file inclusion and macro definitions, because only they were relevant for translating Haskell modules. Description of all supported directives could be found in the C99 standard [6], while [7] 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, it triggers an expansion: it replaces the invocation with the tokens from its body. In this paper the "⇒" notation will represent 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. A macro expansion could never result in a new preprocessing directive, so it is impossible to define a macro by expanding an existing one.

To guarantee 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:

```
#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 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 name of the currently expanding macro is not yet appended to the expansion list. Argument prescan makes preprocessing strict, as each argument will be expanded before it gets substituted. Without this mechanism, the following nested substitutions would not happen:

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

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. 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.

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

As the example shows, argument prescan is not applied to the operands of these operators. For this reason, a second expansion is needed to prescan their arguments:

```
#define PRINT(x) PRINT_(x)
#define CONCAT(x, y) CONCAT_(x, y)
```

`PRINT(ONE)` $\underset{\textit{prescan}}{\Rightarrow}$ `PRINT(1)` $\Rightarrow$ `PRINT_(1)` $\Rightarrow$ `printf("1")`

`CONCAT(ONE, TWO)` $\underset{\textit{prescan}}{\Rightarrow}$ `CONCAT(1, 2)` $\Rightarrow$ `CONCAT_(1, 2)` $\Rightarrow$ `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. Any definition that is exported in its containing Haskell module will be a preprocessor macro with the same name. Import declarations between Haskell modules will be translated into preprocessor include directives. All the exported definitions of a Haskell module are transformed into macros that are usable by including the generated header file. From Haskell modules importing each other the translation creates header files including each other. 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. These similarities enable the use of prewritten macros as imported foreign functions in Haskell code. Invocation to existing macros can be integrated into a new, translated system.

The translation cannot be applied to IO computations, because preprocessor macros cannot perform file modification or network activity. Handling exceptional cases that can be done in an IO calculation in Haskell can be performed when using the macro definitions in a C source file.

Macro definitions in this paper appear in their original syntax. For demonstration purposes, we added meta-syntax 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))
```

3 BASIC TRANSFORMATIONS

3.1 Primary Data and Operations

The transformation described in this paper generates preprocessor macros that use the *Boost Preprocessing library* [8, 9]. Boost is used as a runtime library for the

generated code. 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 two 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. 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.

All of these primitive types and data structures come with limitations. Numbers are limited to the range of $[0, 256]$, tuples can have a maximal length of 64 elements, while sequences can hold at most 256 elements in the current version of Boost.

3.2 Objects in the Macro System

Three kinds of Haskell objects have to be represented by token lists when the system of generated macros is preprocessed. These are values, thunks and exceptions

Values are data structures that represent a Haskell value. Values can be of primitive types or algebraic data types. For example, the number unboxed 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. The parts of the thunk can be accessed by the `TH_NAME`, `TH_REQ_ARGS` and `TH_ARGS` macros. As an example, the function `(+3)` is represented as the following tuple:

```
(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 propagated by the generated code when an exception appears as a function inside a function application or when an exception is the subject of pattern matching. In these cases the result of the expression will be the unchanged exception. 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. The resulting token list will contain parts that are the direct transformation of the Haskell values that were the parts of the original Haskell binding.

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.

Boolean values are 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
```

Unboxed integral values can be represented with numeric tokens. Integral operations can be performed by concatenating these tokens to macro names. The same token concatenation based approach can be used as for Boolean values. Of course, this means that the range of integral values will be limited. For example, see 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
```

For practical reasons, characters should not be represented invidually, only as part of 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 [10] are the building blocks of all complex types in Haskell. In fact, the Boolean type shown above is also an example of algebraic data type as well as the boxed `Int` and `Char` types that are usually taken as primitive types.

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 algebraic data type 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 value `Just 3` is the following:

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

3.4 Storing Textual Information

One of the goals of our system is to provide a high-level abstraction for code generation. To support this we need to find a good representation for text. The conventional way to represent text in Haskell are Strings, that are lists of characters. There are two problems with storing textual information in Strings. First, the preprocessor operates on the level of tokens, therefore white space characters cannot be represented as tokens. Second, representing text with the Haskell Strings also comes with a big overhead because the list data structure contains many constructors. From these constructors a lot of unnecessary preprocessor constructions would be generated, because the text is usually constant or a concatenation.

For these reasons an alternative Haskell type, named *TokenList* is provided for creating large sections of text. The text represented as TokenList is mapped to simple tokens in the generated preprocessor definitions, without any overhead. The construction of TokenLists is limited in Haskell. Text literals can have TokenList type if they contain a balanced number of parentheses and quotes and do not contain the # character. This constraint will be checked at compile-time. It is important to state that a Haskell String cannot be converted into a TokenList, because it would lose its white space information.

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 form a valid token together, for example `"f" ## "()"` is not a valid expression and will result in a runtime error.

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

3.5 Strictness of the Generated Macros

As it is known, the evaluation of a Haskell program is lazy by default [11]. 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 generated macro system emulates lazy evaluation of Haskell, using token concatenation to prevent the preprocessor from eager macro expansion. This is done implicitly, for example, when translating case expressions. When the translation does not use token concatenation, but laziness is required, it can be achieved using the `APPLY_WHEN` macro described later.

3.6 General Transformations Applied

To be able to easily transform expressions and bindings, the structure of the intermediate representation must be altered.

3.6.1 Lambda-Lifting

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 passed as arguments.

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

3.6.2 Closure Conversion

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 see 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 $\texttt{const}_2$ macro is expanded. This problem is solved by explicitly 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 previous example will be transformed to the following:

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

3.6.3 Type Erasure

A GHC Core Cast expression performs a type coercion on the received expression. Because the Haskell type system is not used during the transformation, Cast expressions are simply ignored and only the expression that is affected by Cast is transformed.

Because GHC Core representation contains type abstraction and 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 top-level 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 expanded. Otherwise the argument is added to arguments collected in the thunk.

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

The APPLY macro is a helper macro that adds a new argument to a (partially applied) function. If the function receives all arguments needed (1), it performs the application by calling the macro that is stored in the thunk (2). Otherwise it reconstructs the thunk with an additional argument added (3). The evaluation must be explicitly delayed until it is needed (4).

```
#define APPLY(f,a)
 (1) IF( EQUAL( SEQ_SIZE(SEQ_ADD(TH_ARGS(f),a)), TH_REQ_ARGS(f)), \
        EXPAND( \
      (4) APPLY_WHEN( \
            NOT_EQUAL( SEQ_SIZE(SEQ_ADD(TH_ARGS(f),a)), TH_REQ_ARGS(f)), \
        (2) TH_NAME(f), \
            SEQ_TO_TUPLE(SEQ_ADD(TH_ARGS(f),a))) ) , \
    (3) (THUNK)(TH_NAME(f))(TH_REQ_ARGS(f)) \
              (SEQ_ADD(TH_ARGS(f),a)))
```

The macros used above (except APPLY_WHEN and IF) are Boost macros with simplified names. The APPLY_WHEN macro performs a lazy evaluation, expanding

the given macro only when the condition holds, otherwise it separates the macro name from the argument list with a $ symbol to prevent eager macro expansion by the preprocessor. Eager macro expansion would produce preprocessing errors even if the branch that causes the errors is not selected by the branching macro. This is the reason why the same condition appears twice in the definition of `APPLY`, the tokens where the expansion is prevented with the $ symbol will not be used. ($ was chosen because it cannot form a macro call with the parentheses)

```
#define APPLY_WHEN(p,f,args) f IF(p,,$) args

#define IF(p,t,f) CAT(IF_,p)(t,f)
#define IF_1(t,f) t
#define IF_0(t,f) f
```

A lambda abstraction is transformed into a thunk that expands the macro generated from the body of the abstraction when the argument is given.

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

The list of macros performing pattern matching can be divided into three parts. The first one checks that the object matched on is a value. The second part propagates the argument unchanged if it is an exception. Finally, the third part concatenates the tag 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.

$$\varphi\begin{pmatrix}\text{case}\, e\, \text{of} & p_1 & \to e_1 \\ & \vdots & \\ & p_n & \to e_n \\ & _ & \to e_{def}\end{pmatrix} = \langle\texttt{try}\rangle(\langle\varphi(e)\rangle)$$

```
#define ⟨try⟩(⟨x⟩) IF(IS_EXCEPTION(⟨x⟩), x, ⟨match⟩(x))
#define ⟨match⟩(⟨x⟩) IF( \
  IS_COVERED(TAG(⟨x⟩), ⟨ tag(p1)... tag(pn)⟩), \
  ⟨dispatch⟩(x), \
  APPLY_WHEN(IS_COVERED(TAG(⟨x⟩),⟨ tag(p1)... tag(pn)⟩),⟨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 macro 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 algebraic data type 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 simple maps constructors without parameters to values. It also creates thunks from constructors that have arguments. After enough parameters are given the 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. The constructor of the `Pair` datatype will become a thunk, so it can be curried.

```
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 Problem with Recursive Macros

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 problem applies to the helper macros in our system. For example the `APPLY` macro will be called in the outermost expression and also in the definition of `APP_BODY` when the macros for 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⟩)
  ⇒(prescan) 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, then $\texttt{APPLY}_1$ 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.

4.2 Translating Recursive Functions

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

$$\varphi' \begin{pmatrix} map = \lambda f.\lambda l.\,\mathrm{case}\, l\, \mathrm{of} & \\ \qquad\qquad\qquad \mathrm{nil} \rightarrow \mathrm{nil} & \\ \qquad\qquad \mathrm{cons}\ \ h\ t \rightarrow \mathrm{cons}\ (f\ \ h)\ (map\ \ f\ \ t) & \end{pmatrix} =$$

```
#define CASE_nil() nil
#define CASE_cons(f,h,t) \
        APPLY(cons,APPLY(f,h),APPLY(map,f,t))
#define DISPATCH(l,f) CASE_ ## TAG(l) APPEND(f, ARGS(l))
#define map_BODY(f,l) TRY(DISPATCH,l,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 of 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 has to estimate how many of times his function can appear in a call chain (expansion of nested macros). There is a default number, but it can be overridden by the user as it can be seen in Section 4.4. 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. This approach will also support the use of mutually recursive declarations as long as no higher-order functions are involved.

4.3 Translating Higher-Order Functions

One problem with higher-order functions is that they can receive themselves as arguments. For example, take the 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 for 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 check 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. Depending on the condition `p`, the macro generated from `f` should be expanded multiple times in a single call chain, which will be prevented by the preprocessor.

These problems are similar to the problem with higher-order functions that receive 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.

Fortunately, only a few higher order functions are often used in source code analysis and transformation. They are the simple list processing functions (`map`, `fold`), usually used on reasonably small lists.

4.4 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 evaluated 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 ten 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 the transformation of a given definition. The 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.

```
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.

In both approaches, `Recursive` is defined in hs2cpp. In the first case it is a simple datatype with a numeric field, while in the second case, it is a type class with a numeric parameter.

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

5 EXAMPLE: MEMORY LAYOUT CONFIGURATION

This section presents a macro library that enables to change the memory layout of data in C programs easily, which is useful when working on hardware with different types of memories and programmable caches. On these platforms it is a recurring scenario when a part of a data structure should be modified to be stored in a different memory layer than it was before.

The first step of this transformation is usually the introduction of an indirection level with a pointer in the original structure to allow the relocation of the given data element into a different memory. To represent indirection levels in the memory layout of data elements, *configurations* are introduced. For a given configuration it is possible to automatically derive C declarations, allocation and deallocation statements and expressions to access the configured data. Changing the memory layout of a configurable code then involves changing the configuration only, as any operations working with the data are synthetized from it.

The code listing below shows how the declaration, allocation and an array element access could be achieved with macros. It shows the usage of a `declare` macro for the declaration of an array with eight elements, which are containing pointers to data structures. The allocation code for the data elements will be synthetized by macro `allocate`. Macro `value_at` is used to access the element at index two in the array, and the resulting expression can be used as a left- or right-value in another C expression.

```
#define configuration Array(8, Pointer(Scalar))

declare(struct data, all_data, configuration)
allocate(all_data, configuration)
...value_at(all_data, configuration, 2)...
```

To understand the representation of configurations in depth, consider the following C declarations and their corresponding configuration:

```
struct data sample;       // Scalar
struct data *sample;      // Pointer(Scalar)
struct data sample[8];    // Array(8, Scalar)
struct data *sample[8];   // Array(8, Pointer(Scalar))
struct data (*sample)[8]; // Pointer(Array(8, Scalar))
```

A `Scalar` configuration implies storing a single data element without any indirection. The `Pointer` modifier introduces a single pointer level to its parameter configuration. Multiple data elements can be represented with the `Array` modifier, which contains the number of data elements and their configuration. The innermost item in a configuration is always `Scalar`. `Pointer` and `Array` modifiers can be used on a configuration arbitrary to construct a new configuration. Configurations are easy to implement in Haskell with the following algebraic data type:

```
data Config = Scalar | Pointer Config | Array Int Config
```

The constructors of this data type will be compiled to macros by *hs2cpp*, enabling them to be used from C in exactly the same way as it was shown in the comments above. From this point, any code synthetization macros can be modeled with functions which are processing configurations. For clarity, introduce the following type aliases:

```
type Code = TokenList
type TypeName = TokenList
type VarName = TokenList
```

The `TokenList` type were introduced in Section 3.4 to store text which will be compiled into raw preprocessor tokens by *hs2cpp*. For simplicity `TokenList` is also used in the cases when a single token is created. Using these types it is possible to create a function which synthesizes C declarations from a base type, a variable name and a configuration:

```
declare :: TypeName -> VarName -> Config -> Code
declare baseType var c = baseType # typedName c # ";"
  where
    typedName :: Recursive 10 => Config -> Code
    typedName Scalar = var
    typedName (Pointer (Array d c))
     = paren ("*" # typedName c) # "[" # tokenize d # "]"
    typedName (Pointer c) = "*" # typedName c
```

```
  typedName (Array d c)
   = typedName c # "[" # tokenize d # "]"
```

After compilation into macros, the resulting library can be used in C code as follows:

```
declare(struct data, all_data, Array(8, Pointer(Scalar)))
// expands to: struct data *all_data[8];
```

For the recursive helper functions it is important to specify how deep the recursion could be invoked. As the `Recursive` constraint on the helper function shows, `declare` supports processing of at most nine modifiers on a `Scalar` configuration.

The predefined `tokenize` function is used to convert an integer to a `TokenList` using its `Show` instance. Operator `#` is used to concatenate generated code stored in `TokenList` objects, while `paren` operator puts its argument between parentheses. As the implementation construct token lists from string literals, the language extension `OverloadedStrings` must be turned on.

Allocation and deallocation statements, and data access expression could be implemented in the same way by processing the given configuration.

6 MEASUREMENT RESULTS

In an experiment, we measured the preprocessing time of a manually written preprocessor code and its corresponding automatically generated counterpart. To measure the differences we implemented two versions of the code presented in Section 5. The main functionality was the implementation of `DECLARE` and `VALUE_OF` macros. These can generate the declaration and the evaluation of a configured variable. One of the implementations was written in Haskell, the other was written directly as preprocessor macros using the Boost library.

For translating the Haskell version of the code, we used a maximum recursion limit of 10. The maximum recursion limit of the handwritten version was the limit on Boost quasi-recursive macros, that is 64. Each measurement was performed in five cases with different amount of declarations needed to be generated in each case the preprocessing was repeated 100 times and the amount of time taken was averaged. The preprocessor shipped with gcc 5.2.0 was used, on a ThinkPad T440p laptop with Intel i7 processor and 16 GB RAM. The results are presented in Table 2.

	$1\times$	$2\times$	$4\times$	$8\times$	$16\times$
Handwritten	86	120	215	526	1 730
Generated	1 945	3 819	7 777	16 136	34 888

Table 2. This table shows the amount of time (in milliseconds) taken to run the preprocessor in each case with the handwritten and generated code

Although the performance difference is an order of magnitude, it is worth noticing that while the generated preprocessor macros are not optimized, the handwritten

ones are implemented using the already optimized Boost macros. This difference may account for the most of this large difference in performance.

7 RELATED WORK

As mentioned earlier, the *Boost Preprocessing library* [8] provides macros for handling various data structures, including tuples, arrays, lists and sequences. Operations on Boolean and integral values, like negation, addition and subtraction in limited ranges are also supported. It also provides emulation of conditional and loop control structures. Some list-processing operations, like filter, map and fold are also supported. This also shows that operations familiar from functional languages 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 [12]. In [13], Zoltán Porkoláb describes the connection between functional programs and C++ templates. There is a chapter about embedding Haskell into C++ [14] and the conversion of these embedded functional programs using the internal representation of YHC, the York Haskell Compiler [15]. 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.

A different solution for generating C++ template metaprograms can be found in [16]. Here the authors used the metaprogramming facilities provided by a functional programming language Racket to support EDSL development in C++.

The *Boost library* also includes a template metaprogramming library, called *MPL* [9]. 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 allowed to be recursive. However, it is limited to C++ and cannot generate arbitrary code.

Starting from the C++ 11 standard [17], the *constexpr* specifier allows compile-time usage of variables and functions [18]. 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 compile-time programming method.

Another approach to support metaprogramming in C is to extend the language itself. Meta-C [19] introduces new programming elements into the language while it remains fully backwards compatible with the C99 standard [6] it is 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 [20], the last release was an alpha version in 2007.

Other macro languages have semantics that are different. For example the M4 preprocessor [21] 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

The main contribution of this paper is a method to automatically translate Haskell programs into C preprocessor directives. 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 features and 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 our implemented solutions for each of these problems, although for recursion and higher-order functions these solutions have some limitations.

Solutions for special cases of higher-order and recursive functions were provided in the Section 4 along with their limitations. Generating reliable preprocessor code for recursive and higher-order functions without serious limitations requires further research.

We compared the performance of the generated macros to handwritten ones in Section 6. We found that although handwritten preprocessor macros are faster to preprocess, the difference could be reduced by optimization of the generated code.

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

REFERENCES

[1] Karácsony, M.: Modeling C Preprocessor Metaprograms Using Purely Functional Languages. Proceedings of the 9$^{\text{th}}$ International Conference on Applied Informatics, 2014, Vol. 2, pp. 85–92.

[2] Marlow, S.—Peyton Jones, S. et al.: The Glasgow Haskell Compiler. 2004.

[3] Peyton Jones, S. L.—Hall, C.—Hammond, K.—Cordy, J.—Kevin, H.—Partain, W.—Wadler, P.: The Glasgow Haskell Compiler: A Technical Overview. 1992.

[4] Sulzmann, M.—Chakravarty, M. M. T.—Peyton Jones, S.—Donnelly, K.: System F with Type Equality Coercions. Proceedings of the 2007 ACM SIGPLAN

International Workshop on Types in Languages Design and Implementation, ACM, 2007, pp. 53–66.

[5] Pierce, B. C.: Types and Programming Languages. MIT, 2002, pp. 341–344.

[6] ISO/IEC JTC1/SC22/WG14. Programming Languages – C. Standard, International Organization for Standardization, Vol. 12, 1999.

[7] Garrido, A.—Meseguer, J.—Johnson, R.: Algebraic Semantics of the C Preprocessor and Correctness of Its Refactorings. 2006.

[8] Boost Preprocessor Library. Accessed: February 20, 2015.

[9] Abrahams, D.—Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond. Pearson Education, 2004.

[10] Peyton Jones, S.—Wadler, P.—Hudak, P.—Hughes, J.: A History of Haskell: Being Lazy with Class. Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), 2007, pp. 14–15.

[11] Marlow, S.: Haskell 2010 Language Report.

[12] Veldhuizen, T. L.: C++ Templates are Turing Complete. Available on Citeseer. ist. Psu. edu/581150. HTML, 2003.

[13] Porkoláb, Z: Functional Programming with C++ Template Metaprograms. In: Horváth, Z., Plasmeijer, R., Zsók, V. (Eds.): Central European Functional Programming School. Springer Berlin Heidelberg, Lecture Notes in Computer Science, Vol. 6299, 2010, pp. 306–353.

[14] Porkoláb, Z.—Sinkovics, Á.: C++ Template Metaprogramming with Embedded Haskell. Proceedings 8th International Conference Generative Programming and Component Engineering (GPCE 2009), New York, NY, USA, ACM, 2009, pp. 99–108.

[15] York Haskell Compiler. `https://wiki.haskell.org/Yhc`. Accessed: February 20, 2015.

[16] Ballantyne, M.—Earl, Ch.—Might, M.: Meta-Meta-Programming. 2014 Scheme and Functional Programming Workshop, 2015.

[17] ISO/IEC JTC1/SC22/WG21. Programming Languages – C++. Standard, International Organization for Standardization, Vol. 09, 2011.

[18] Stroustrup, B.: Programming: Principles and Practice Using C++. Pearson Education, 2014.

[19] Maier-Komor, T.—Färber, G.: MetaC: A Metaprogramming Extension to C Enabling Crosscutting Reconfiguration of Embedded Software.

[20] The MetaC Language. `http://www.maier-komor.de/metac/`. Accessed: February 20, 2015.

[21] M4 Preprocessor Language. `http://wolfram.schneider.org/bsd/7thEdManVol2/m4/m4.pdf`. Accessed: February 20, 2015.

9 APPENDIX

In this section a complete example is presented that shows the transformation from the original Haskell program to the final results of the preprocessing. Section 9.1 shows the original Haskell program with a type class, a data declaration and two

functions. Section 9.2 shows the macros generated from the Haskell code, while Section 9.3 shows the helper macros generated for the evaluation. Finally, Section 9.4 shows the suggested usage of the macros that result shown in Section 9.5.

9.1 Haskell Code

```
{-# LANGUAGE OverloadedStrings, MagicHash #-}
-- exported definitions can be used
module Demo (mark, sumSqr) where

import GHC.Base (Int(..), (+#), (*#))
import Prelude hiding (Num, (+), (*))
import HaskellToMacro.Prelude

-- Must redefine classes like Num in order to be able
-- to use them.
infixl 5 +
infixl 6 *
class  Num a  where
    (+), (*) :: a -> a -> a
    -- ...

instance  Num Int  where
    I# x + I# y = I# (x +# y)
    I# x * I# y = I# (x *# y)
    -- ...

-- Class instances are resolved
sumSqr :: Int -> Int -> Int
sumSqr a b = a * a + b * b

-- The defined data types can be used
data Mark = A | B | C | D | E

mark :: Mark -> TokenList
mark A = "a"
mark B = "b"
mark C = "c"
-- explicit errors are handled
mark D = error "cannot give a d"
-- error is generated if E is given
```

9.2 Macros Generated from Haskell Code

The code generated from the `Num` type class:

```
#define HS2CPP_CASE_ALT_2_1(_doQ,_doR,_ann,_ano)
    ↪ HS2CPP_APPLY_3(_6d,HS2CPP_APPLY_3(HS2CPP_APPLY_3((
    ↪ HS2CPP_THUNK)(HS2CPP_MUL)(2)(),_ann),_ano))
#define HS2CPP_CASE_DISPATCH_3(_doQ,_doR,_ann,t)
    ↪ BOOST_PP_CAT(HS2CPP_CASE_ALT_2_,BOOST_PP_TUPLE_ELEM
    ↪ (2,0,BOOST_PP_SEQ_ELEM(1,t)))BOOST_PP_SEQ_TO_TUPLE
    ↪ ((_doQ)(_doR)(_ann)BOOST_PP_TUPLE_ELEM(2,1,
    ↪ BOOST_PP_SEQ_ELEM(1,t)))
#define HS2CPP_CASE_ALT_1_1(_doQ,_doR,_ann)
    ↪ HS2CPP_TRY_CTX_2(HS2CPP_CASE_DISPATCH_3,_doR,_doQ,
    ↪ _doR,_ann)
#define HS2CPP_CASE_DISPATCH_4(_doQ,_doR,t) BOOST_PP_CAT(
    ↪ HS2CPP_CASE_ALT_1_,BOOST_PP_TUPLE_ELEM(2,0,
    ↪ BOOST_PP_SEQ_ELEM(1,t)))BOOST_PP_SEQ_TO_TUPLE((_doQ
    ↪ )(_doR)BOOST_PP_TUPLE_ELEM(2,1,BOOST_PP_SEQ_ELEM(1,
    ↪ t)))
#define HS2CPP_LAM_BODY_0(_doQ,_doR) HS2CPP_TRY_CTX_1(
    ↪ HS2CPP_CASE_DISPATCH_4,_doQ,_doQ,_doR)
#define _aot (HS2CPP_THUNK)(HS2CPP_LAM_BODY_0)(2)()
// $c*[_aot]: Int -> Int -> Int
#define _36c_42(_sp3,_sp5) HS2CPP_IF_0(
    ↪ HS2CPP_IS_EXCEPTION(HS2CPP_APPLY_0(HS2CPP_APPLY_0(
    ↪ _aot,(HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_sp3))))),(
    ↪ HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_sp5)))))),
    ↪ HS2CPP_APPLY_0(HS2CPP_APPLY_0(_aot,(HS2CPP_VALUE)
    ↪ ((1,((HS2CPP_VALUE)(_sp3))))),(HS2CPP_VALUE)((1,((
    ↪ HS2CPP_VALUE)(_sp5))))),(BOOST_PP_SEQ_ELEM(0,
    ↪ HS2CPP_APPLY_0(HS2CPP_APPLY_0(_aot,(HS2CPP_VALUE)
    ↪ ((1,((HS2CPP_VALUE)(_sp3))))),(HS2CPP_VALUE)((1,((
    ↪ HS2CPP_VALUE)(_sp5)))))))(BOOST_PP_SEQ_ELEM(1,
    ↪ BOOST_PP_SEQ_ELEM(0,BOOST_PP_TUPLE_ELEM(2,1,
    ↪ BOOST_PP_SEQ_ELEM(1,HS2CPP_APPLY_0(HS2CPP_APPLY_0(
    ↪ _aot,(HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_sp3))))),(
    ↪ HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_sp5))))))))))

#define HS2CPP_CASE_ALT_7_1(_doO,_doP,_anl,_anm)
    ↪ HS2CPP_APPLY_8(_6d,HS2CPP_APPLY_8(HS2CPP_APPLY_8((
    ↪ HS2CPP_THUNK)(HS2CPP_ADD)(2)(),_anl),_anm))
#define HS2CPP_CASE_DISPATCH_8(_doO,_doP,_anl,t)
    ↪ BOOST_PP_CAT(HS2CPP_CASE_ALT_7_,BOOST_PP_TUPLE_ELEM
```

```
    ↪ (2,0,BOOST_PP_SEQ_ELEM(1,t)))BOOST_PP_SEQ_TO_TUPLE
    ↪ ((_doO)(_doP)(_anl)BOOST_PP_TUPLE_ELEM(2,1,
    ↪ BOOST_PP_SEQ_ELEM(1,t)))
#define HS2CPP_CASE_ALT_6_1(_doO,_doP,_anl)
    ↪ HS2CPP_TRY_CTX_7(HS2CPP_CASE_DISPATCH_8,_doP,_doO,
    ↪ _doP,_anl)
#define HS2CPP_CASE_DISPATCH_9(_doO,_doP,t) BOOST_PP_CAT(
    ↪ HS2CPP_CASE_ALT_6_,BOOST_PP_TUPLE_ELEM(2,0,
    ↪ BOOST_PP_SEQ_ELEM(1,t)))BOOST_PP_SEQ_TO_TUPLE((_doO
    ↪ )(_doP)BOOST_PP_TUPLE_ELEM(2,1,BOOST_PP_SEQ_ELEM(1,
    ↪ t)))
#define HS2CPP_LAM_BODY_5(_doO,_doP) HS2CPP_TRY_CTX_6(
    ↪ HS2CPP_CASE_DISPATCH_9,_doO,_doO,_doP)
#define _aoq (HS2CPP_THUNK)(HS2CPP_LAM_BODY_5)(2)()
// $c+[_aoq]: Int -> Int -> Int
#define _36c_43(_sp7,_sp9) HS2CPP_IF_5(
    ↪ HS2CPP_IS_EXCEPTION(HS2CPP_APPLY_5(HS2CPP_APPLY_5(
    ↪ _aoq,(HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_sp7))))),(
    ↪ HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_sp9)))))),
    ↪ HS2CPP_APPLY_5(HS2CPP_APPLY_5(_aoq,(HS2CPP_VALUE)
    ↪ ((1,((HS2CPP_VALUE)(_sp7))))),(HS2CPP_VALUE)((1,((
    ↪ HS2CPP_VALUE)(_sp9))))),(BOOST_PP_SEQ_ELEM(0,
    ↪ HS2CPP_APPLY_5(HS2CPP_APPLY_5(_aoq,(HS2CPP_VALUE)
    ↪ ((1,((HS2CPP_VALUE)(_sp7))))),(HS2CPP_VALUE)((1,((
    ↪ HS2CPP_VALUE)(_sp9)))))))(BOOST_PP_SEQ_ELEM(1,
    ↪ BOOST_PP_SEQ_ELEM(0,BOOST_PP_TUPLE_ELEM(2,1,
    ↪ BOOST_PP_SEQ_ELEM(1,HS2CPP_APPLY_5(HS2CPP_APPLY_5(
    ↪ _aoq,(HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_sp7))))),(
    ↪ HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_sp9))))))))))

#define _r0 HS2CPP_APPLY_10(HS2CPP_APPLY_10(_rnr,_aoq),
    ↪ _aot)
// $fNumInt[_r0]: Num Int
#define _36fNumInt _r0

#define HS2CPP_CASE_ALT_17_1(_soT,_soW,_00) _soW
#define HS2CPP_CASE_DISPATCH_18(_soT,t) BOOST_PP_CAT(
    ↪ HS2CPP_CASE_ALT_17_,BOOST_PP_TUPLE_ELEM(2,0,
    ↪ BOOST_PP_SEQ_ELEM(1,t)))BOOST_PP_SEQ_TO_TUPLE((_soT
    ↪ )BOOST_PP_TUPLE_ELEM(2,1,BOOST_PP_SEQ_ELEM(1,t)))
#define HS2CPP_LAM_BODY_16(_soT) HS2CPP_TRY_CTX_19(
    ↪ HS2CPP_CASE_DISPATCH_18,_soT,_soT)
#define _rmB (HS2CPP_THUNK)(HS2CPP_LAM_BODY_16)(1)()
// +[_rmB]: forall a. Num a => a -> a -> a
```

```
#define _43 _rmB

#define HS2CPP_CASE_ALT_20_1(_soY,_00,_sp1) _sp1
#define HS2CPP_CASE_DISPATCH_21(_soY,t) BOOST_PP_CAT(
    ↪ HS2CPP_CASE_ALT_20_,BOOST_PP_TUPLE_ELEM(2,0,
    ↪ BOOST_PP_SEQ_ELEM(1,t)))BOOST_PP_SEQ_TO_TUPLE((_soY
    ↪ )BOOST_PP_TUPLE_ELEM(2,1,BOOST_PP_SEQ_ELEM(1,t)))
#define HS2CPP_LAM_BODY_19(_soY) HS2CPP_TRY_CTX_22(
    ↪ HS2CPP_CASE_DISPATCH_21,_soY,_soY)
#define _rmC (HS2CPP_THUNK)(HS2CPP_LAM_BODY_19)(1)()
// *[_rmC]: forall a. Num a => a -> a -> a
#define _42 _rmC
```

The macros generated from the `sumSqr` function:

```
#define HS2CPP_LAM_BODY_10(_ani,_anj) HS2CPP_APPLY_12(
    ↪ HS2CPP_APPLY_12(HS2CPP_APPLY_12(_rmB,_r0),
    ↪ HS2CPP_APPLY_12(HS2CPP_APPLY_12(HS2CPP_APPLY_12(
    ↪ _rmC,_r0),_ani),_ani)),HS2CPP_APPLY_12(
    ↪ HS2CPP_APPLY_12(HS2CPP_APPLY_12(_rmC,_r0),_anj),
    ↪ _anj))
#define _r4 (HS2CPP_THUNK)(HS2CPP_LAM_BODY_10)(2)()
// sumSqr[_r4]: Int -> Int -> Int
#define sumSqr(_spb,_spd) HS2CPP_IF_11(
    ↪ HS2CPP_IS_EXCEPTION(HS2CPP_APPLY_11(HS2CPP_APPLY_11
    ↪ (_r4,(HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_spb))))),(
    ↪ HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_spd)))))),
    ↪ HS2CPP_APPLY_11(HS2CPP_APPLY_11(_r4,(HS2CPP_VALUE)
    ↪ ((1,((HS2CPP_VALUE)(_spb))))),(HS2CPP_VALUE)((1,((
    ↪ HS2CPP_VALUE)(_spd))))),(BOOST_PP_SEQ_ELEM(0,
    ↪ HS2CPP_APPLY_11(HS2CPP_APPLY_11(_r4,(HS2CPP_VALUE)
    ↪ ((1,((HS2CPP_VALUE)(_spb))))),(HS2CPP_VALUE)((1,((
    ↪ HS2CPP_VALUE)(_spd)))))))(BOOST_PP_SEQ_ELEM(1,
    ↪ BOOST_PP_SEQ_ELEM(0,BOOST_PP_TUPLE_ELEM(2,1,
    ↪ BOOST_PP_SEQ_ELEM(1,HS2CPP_APPLY_11(HS2CPP_APPLY_11
    ↪ (_r4,(HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_spb))))),(
    ↪ HS2CPP_VALUE)((1,((HS2CPP_VALUE)(_spd))))))))))
```

The representation of the `Mark` datatype:

```
// A[_rnv]: Mark
#define A _rnv
#define _rnv (HS2CPP_VALUE)((1,))
// B[_rny]: Mark
#define B _rny
#define _rny (HS2CPP_VALUE)((2,))
```

```
// C[_rnB]: Mark
#define C _rnB
#define _rnB (HS2CPP_VALUE)((3,))
// D[_rnE]: Mark
#define D _rnE
#define _rnE (HS2CPP_VALUE)((4,))
// E[_rnH]: Mark
#define E _rnH
#define _rnH (HS2CPP_VALUE)((5,))
// D_58Num[_rnr]: forall a. (a -> a -> a) -> (a -> a -> a
    ↪ ) -> Num a
#define D_58Num _rnr
#define HS2CPP_CTOR_D_58Num_22(a1,a2) (HS2CPP_VALUE)((1,(
    ↪ a1)(a2)))
#define _rnr (HS2CPP_THUNK)(HS2CPP_CTOR_D_58Num_22)(2)()
```

The code generated from the `mark` function:

```
#define HS2CPP_CASE_ALT_12_1(_doL) (HS2CPP_VALUE)((a))
#define HS2CPP_CASE_ALT_12_2(_doL) (HS2CPP_VALUE)((b))
#define HS2CPP_CASE_ALT_12_3(_doL) (HS2CPP_VALUE)((c))
#define HS2CPP_CASE_ALT_12_4(_doL) (HS2CPP_EXCEPTION)(
    ↪ cannot give a d)
#define HS2CPP_CASE_DISPATCH_13(_doL,t) BOOST_PP_CAT(
    ↪ HS2CPP_CASE_ALT_12_,BOOST_PP_TUPLE_ELEM(2,0,
    ↪ BOOST_PP_SEQ_ELEM(1,t)))BOOST_PP_SEQ_TO_TUPLE((_doL
    ↪ )BOOST_PP_TUPLE_ELEM(2,1,BOOST_PP_SEQ_ELEM(1,t)))
#define HS2CPP_CASE_DEF_CHECK_14(_doL,t) HS2CPP_IF_15(
    ↪ HS2CPP_LIST_MEMBER(BOOST_PP_TUPLE_ELEM(2,0,
    ↪ BOOST_PP_SEQ_ELEM(1,t)),(1,(2,(3,(4,BOOST_PP_NIL)))
    ↪ )),HS2CPP_CASE_DISPATCH_13(_doL,t),HS2CPP_APP_WHEN(
    ↪ BOOST_PP_NOT(HS2CPP_LIST_MEMBER(BOOST_PP_TUPLE_ELEM
    ↪ (2,0,BOOST_PP_SEQ_ELEM(1,t)),(1,(2,(3,(4,
    ↪ BOOST_PP_NIL)))))),HS2CPP_CASE_DEF_EXEC_15,(_doL)))
#define HS2CPP_CASE_DEF_EXEC_15(_doL) (HS2CPP_EXCEPTION)(
    ↪ Demo.hs:(29,1)-(33,32)|function mark)
#define HS2CPP_LAM_BODY_11(_doL) HS2CPP_TRY_CTX_15(
    ↪ HS2CPP_CASE_DEF_CHECK_14,_doL,_doL)
#define _r5 (HS2CPP_THUNK)(HS2CPP_LAM_BODY_11)(1)()
// mark[_r5]: Mark -> TokenList
#define mark(_spf) HS2CPP_IF_14(HS2CPP_IS_EXCEPTION(
    ↪ HS2CPP_APPLY_14(_r5,_spf)),HS2CPP_APPLY_14(_r5,_spf
    ↪ ),(BOOST_PP_SEQ_ELEM(0,HS2CPP_APPLY_14(_r5,_spf)))(
    ↪ HS2CPP_REMOVE_PAREN(BOOST_PP_SEQ_ELEM(1,
    ↪ HS2CPP_APPLY_14(_r5,_spf)))))
```

Data types generated from Haskell standard library:

```
// I_35[_6d]: Int# -> Int
#define I_35(_spn) HS2CPP_IF_0(HS2CPP_IS_EXCEPTION(
    ↪ HS2CPP_APPLY_0(_6d,(HS2CPP_VALUE)(_spn))),
    ↪ HS2CPP_APPLY_0(_6d,(HS2CPP_VALUE)(_spn)),(
    ↪ BOOST_PP_SEQ_ELEM(0,HS2CPP_APPLY_0(_6d,(
    ↪ HS2CPP_VALUE)(_spn))))(BOOST_PP_SEQ_ELEM(1,
    ↪ BOOST_PP_SEQ_ELEM(0,BOOST_PP_TUPLE_ELEM(2,1,
    ↪ BOOST_PP_SEQ_ELEM(1,HS2CPP_APPLY_0(_6d,(
    ↪ HS2CPP_VALUE)(_spn))))))))
#define HS2CPP_CTOR_I_35_26(a1) (HS2CPP_VALUE)((1,(a1)))
#define _6d (HS2CPP_THUNK)(HS2CPP_CTOR_I_35_26)(1)()
```

9.3 Helper Functions

For the sake of simplicity the repetitive helper macros are only presented once.

```
#define HS2CPP_APPLY_0(f,a) HS2CPP_IF_0(BOOST_PP_EQUAL(
    ↪ BOOST_PP_SEQ_SIZE(BOOST_PP_SEQ_PUSH_BACK(
    ↪ BOOST_PP_SEQ_ELEM(3,f),a)),BOOST_PP_SEQ_ELEM(2,f)),
    ↪ HS2CPP_EXPAND_0(HS2CPP_APP_WHEN(BOOST_PP_EQUAL(
    ↪ BOOST_PP_SEQ_ELEM(2,f),BOOST_PP_SEQ_SIZE(
    ↪ BOOST_PP_SEQ_PUSH_BACK(BOOST_PP_SEQ_ELEM(3,f),a))),
    ↪ BOOST_PP_SEQ_ELEM(1,f),BOOST_PP_SEQ_TO_TUPLE(
    ↪ BOOST_PP_SEQ_PUSH_BACK(BOOST_PP_SEQ_ELEM(3,f),a))))
    ↪ ,(HS2CPP_THUNK)(BOOST_PP_SEQ_ELEM(1,f))(
    ↪ BOOST_PP_SEQ_ELEM(2,f))(BOOST_PP_SEQ_PUSH_BACK(
    ↪ BOOST_PP_SEQ_ELEM(3,f),a)))
#define HS2CPP_EXPAND_0(t) t
#define HS2CPP_IF_0(p,t,f) BOOST_PP_CAT(HS2CPP_IF_0_,p)(t
    ↪ ,f)
#define HS2CPP_IF_0_1(t,f) t
#define HS2CPP_IF_0_0(t,f) f
#define HS2CPP_TRY_CTX_0(f,t,r...) HS2CPP_IF_7(
    ↪ HS2CPP_IS_EXCEPTION(t),t,f(r,t))

#define HS2CPP_APP_WHEN(p,f,args) f HS2CPP_EXPR_IF_0(
    ↪ BOOST_PP_NOT(p),$)args
#define HS2CPP_LIST_MEMBER(e,l) BOOST_PP_LIST_IS_CONS(
    ↪ BOOST_PP_LIST_FILTER(HS2CPP_LIST_MEMBER_PRED,e,l))
#define HS2CPP_LIST_MEMBER_PRED(d,data,elem)
    ↪ BOOST_PP_EQUAL(data,elem)
#define HS2CPP_ADD(a,b) (HS2CPP_VALUE)(BOOST_PP_ADD(
    ↪ BOOST_PP_SEQ_ELEM(1,a),BOOST_PP_SEQ_ELEM(1,b)))
```

```
#define HS2CPP_MUL(a,b) (HS2CPP_VALUE)(BOOST_PP_MUL(
    ↪ BOOST_PP_SEQ_ELEM(1,a),BOOST_PP_SEQ_ELEM(1,b)))
#define HS2CPP_IS_EXCEPTION(o) BOOST_PP_EQUAL(
    ↪ BOOST_PP_SEQ_ELEM(0,o),HS2CPP_EXCEPTION)
#define HS2CPP_UNWRAP(o) BOOST_PP_SEQ_ELEM(1,o)
#define HS2CPP_VALUE 0
#define HS2CPP_THUNK 1
#define HS2CPP_EXCEPTION 2
#define HS2CPP_REMOVE_PAREN(x...) HS2CPP_EVAL(
    ↪ HS2CPP_REMOVE_PAREN_II x)
#define HS2CPP_EVAL(x...) x
#define HS2CPP_REMOVE_PAREN_II(x...) x
```

9.4 Usage

The macro `HS2CPP_IS_EXCEPTION` checks if the evaluation of a given macro results in an exception. `HS2CPP_UNWRAP` lets the user access the actual result or the error message. (See Section 3.2 for the representation of values and exceptions.) This checked usage is only necessary if it is possible that the evaluation will raise an exception.

```
#define markA mark(A)
#if HS2CPP_IS_EXCEPTION(markA)
#pragma message(HS2CPP_UNWRAP(markA))
#error "Error while expanding HS2CPP-generated macro:
  mark(A)."
#else
int HS2CPP_UNWRAP(markA) = 3;
#endif

#define markD mark(D)
#if HS2CPP_IS_EXCEPTION(markD)
#pragma message(HS2CPP_UNWRAP(markD))
#error "Error while expanding HS2CPP-generated macro:
  mark(D)."
#else
int HS2CPP_UNWRAP(markD) = 4;
#endif

#define markE mark(E)
#if HS2CPP_IS_EXCEPTION(markE)
#pragma message(HS2CPP_UNWRAP(markE))
#error "Error while expanding HS2CPP-generated macro:
  mark(E)."
```

```
#else
int HS2CPP_UNWRAP(markE) = 5;
#endif

#define sumSqrOneTwo sumSqr(1,2
#if HS2CPP_IS_EXCEPTION(sumSqrOneTwo)
#pragma message(HS2CPP_UNWRAP(sumSqrOneTwo))
#error "Error while expanding HS2CPP-generated macro:
  sumSqr(1,2)."
#else
int preCalc = HS2CPP_UNWRAP(sumSqrOneTwo);
#endif
```

9.5 Result of Preprocessing

```
int a = 3;

#pragma message(cannot give a d)
Demo.h:11729:2: error: #error "Error while expanding
    ↪ HS2CPP-generated macro: mark(D)."

#pragma message(Demo.hs:(29,1)-(33,32)|function mark)
Demo.h:11736:2: error: #error "Error while expanding
    ↪ HS2CPP-generated macro: mark(E)."

int preCalc = 5;
```

Boldizsár Németh works as researcher at the Eötvös Loránd University. He worked on projects ranging from functional programming to assembly programing. He holds lectures on Java and compilers. He has interest in domain specific languages, modeling and development of programming tools.

Máté Karácsony is Ph.D. student at the Eötvös Loránd University. He is interested in any aspects of source-to-source transformations, including refactoring, source-level optimization and transpilation. He worked on different projects involving also domain-specific and functional languages.

Zoltán Kelemen is a student at the Eötvös Loránd University and a software engineer at Ericsson. His research areas are functional and concurrent programming, domain-specific and embedded languages and compilers.

Máté Tejfel is Assistant Professor at the Eötvös Loránd University, Budapest. He received his Ph.D. degree in computer science in 2009. His research fields are refactoring, functional programming, software verification and domain specific languages.