hs2cpp: Defining C Preprocessor Macro Libraries with Functional Programs

Boldizsár Németh, Máté Karácsony,
Zoltán Kelemen, Máté Tejfel

Eötvös Loránd University, Budapest

{nboldi,kmate,kelemzol,matej}@elte.hu

February 18, 2016

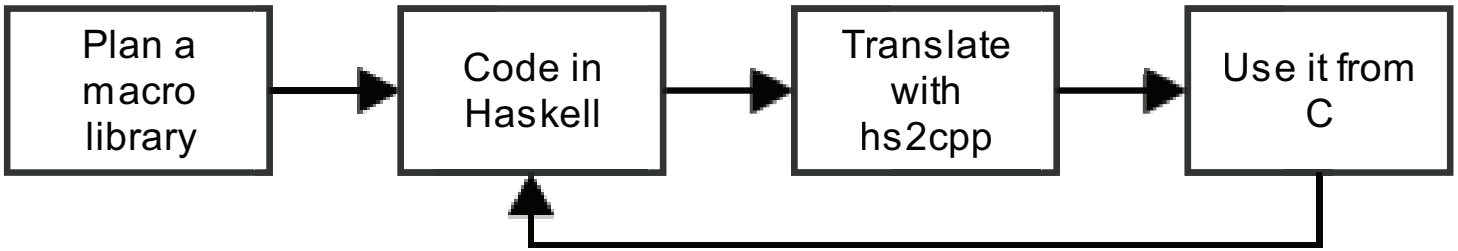
Haskell $\Rightarrow$ C macros (`#define`, ...)

WHY?

HOW?

Motivation

- C macros are often use to extend the language
  - Compile-time reflection
  - Compile-time code generation
  - Deriving related constants
  - Supporting multiple architectures, libraries, platforms
  - Prototyping source-to-source transformations
- Difficult to read, write and maintain them (even with *Boost PP*)
- New workflow:



Preprocessor

- Untyped
- No side-effects
- Branching with token pasting

```
#define IF(x) IF_ ## x
#define IF_TRUE foo
#define IF_FALSE bar

IF(TRUE) ⇒ foo
IF(FALSE) ⇒ bar
```

- No recursion allowed

```
#define REC(x) x + REC(x)

REC(1) ⇒ 1 + REC(1) ≠ 1 + 1 + 1 + ...
```

Motivating example

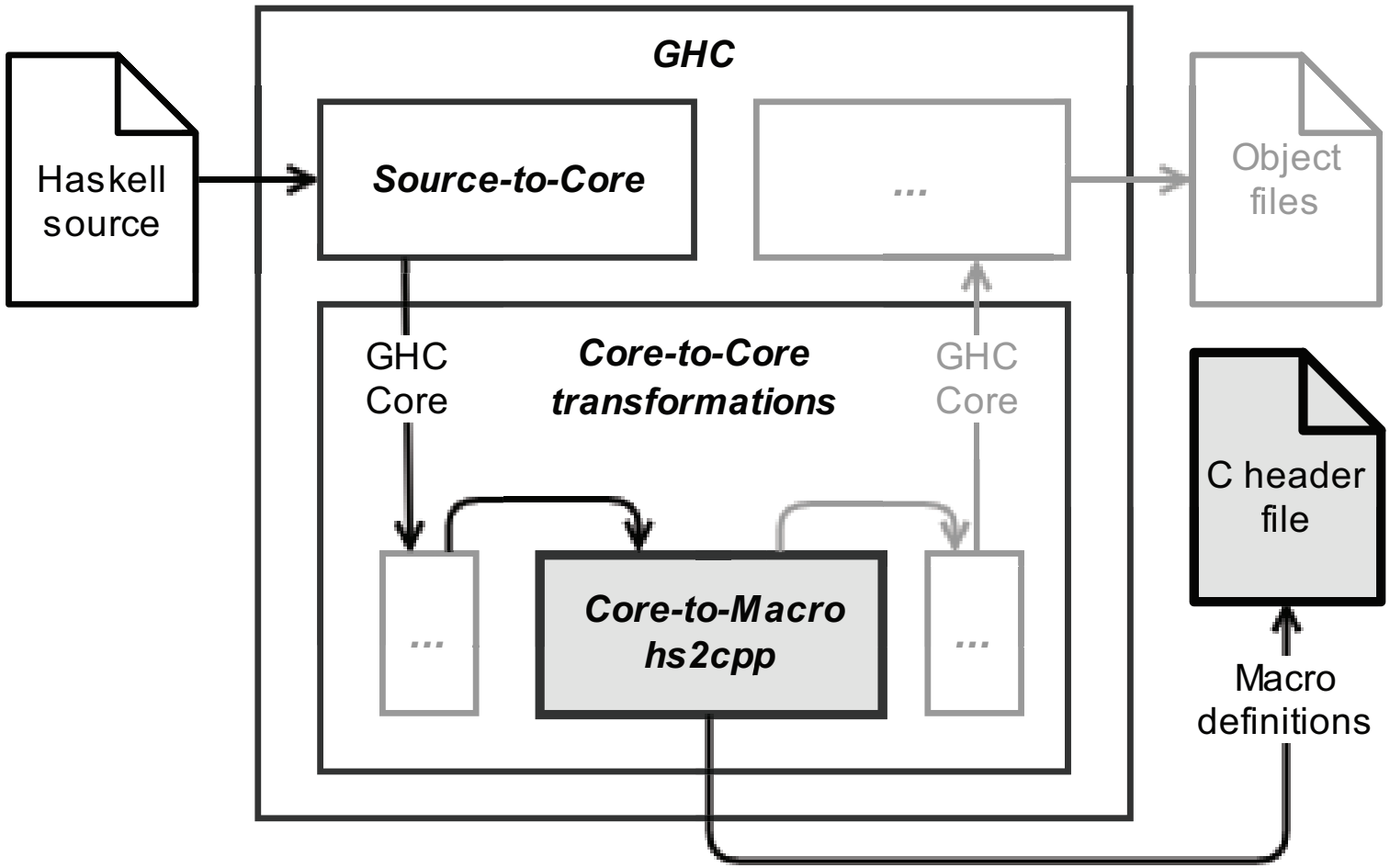
Haskell

```
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 # "]"
```

Handwritten

```
#if NAME_BASED_CONFIGURATION < 1
#define DECLARE DECLARE_IMPL
#else
#define DECLARE(base_type, id) \
    DECLARE_IMPL(base_type, id, GET_OBJECT_CONFIG(id))
#endif

#define DECLARE_IMPL(base_type, id, c) IIF(\
    GREATER(ARRAY_RANK(c), 0), \
    ERROR(For_array_configurations_use_the_DECLARE_ARRAY_macro), \
    base_type FST(DECLARE_(id, c)) )

#define DECLARE_(id, c) FOLDL(DECLARE_APPLY_TOKEN, (EMPTY, id), c)
#define DECLARE_APPLY_TOKEN(s, state, token) \
   CAT(DECLARE_APPLY_TOKEN_, token) state
#define DECLARE_APPLY_TOKEN_SCALAR(code, id) (code id, id)
#define DECLARE_APPLY_TOKEN_PTR(code, id)    (*code, id)

#define ARRAY_RANK(c) FOLDL(ARRAY_RANK_, 0, c)
#define ARRAY_RANK_(s, state, token) CAT(ARRAY_RANK_, token) (state)
#define ARRAY_RANK_SCALAR(cd) cd
#define ARRAY_RANK_PTR(cd)    cd
#define ARRAY_RANK_ARRAY(cd)  INC(cd)

#define IIF           BOOST_PP_IIF
#define FOLDL         BOOST_PP_SEQ_FOLD_LEFT
#define GREATER       BOOST_PP_GREATER
#define CAT           BOOST_PP_CAT
#define FST(tuple)    BOOST_PP_TUPLE_ELEM(0, tuple)
```

Architecture



Transformation I.

- Integers:

```
42                          (VALUE)(42)
```

- Exceptions:

```
error "error message"       (EXCEPTION)((error message))
```

- Functions:

```
\x -> x + 1                 (THUNK)(BODY)(1)()
                            #define BODY(x) PLUS(x,1)
```

- Function application:

```
f 3                         APPLY(f,(VALUE)(3))
```

Transformation II.

- Variables: Using GHC's unique names

```
id x = x
```

```
#define id \
  (THUNK)(BODY)(1)()
#define BODY(_xyz212) _xyz212
```

- Algebraic data types:

```
data List a
  = Nil
  | Cons a List
```

```
#define Nil \
  (VALUE)((NilTag))
#define Cons \
  (THUNK)(ConsBody)(2)()
#define ConsBody(h,t) \
  (VALUE)((ConsTag,(h)(t)))
```

Pattern matching

Steps of pattern matching

- Test for exception
- Test for default
- Select the correct case

```
case x of
  1 -> 3
  2 -> 4
  _ -> 5
```

```
IF(EXCEPTION(x), x, MATCH(x))
#define MATCH(x) \
  IF(COVERED(TAG(x), (tag_1)(tag_2)), \
     DISPATCH, DEFAULT)
#define DISPATCH(x) \
     p_ ## TAG(x)ARGS(x)
#define p_1(x) (VALUE)(3)
#define p_2(x) (VALUE)(4)
#define DEFAULT (VALUE)(5)
```

Scoping issues

- Lambda lifting:

```
f p = let a = p
      in a + 12
```

```
f p = (a p) + 12
a p' = p'
```

- Closure conversion:

```
const
  = \a ->
      \b -> a
```

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

Text

- Representing text as Strings (character lists)
  - loosing whitespaces
  - inefficient
- Solution:
  - a new `TokenStream` type in Haskell
  - store textual data as raw tokens

```
brackets :: TokenStream
         -> TokenStream
brackets s
  = "[" # s # "]"
```

```
#define brackets \
  (TRUNK)(BODY)(1)()
#define BODY(s) \
  CONCAT((VALUE)([),
    CONCAT(s,(VALUE)(])))
```

Recursion

- No recursion in the preprocessor
- Simulated for a given depth
- The recursion limit set by type annotation

```
replicate :: Recursive 10
          => TokenStream
          -> TokenStream
replicate s
  = s # replicate s
```

```
#define REPLICATE_0 \
  (THUNK)(BODY_0)(1)()
#define BODY_0(s) \
  CONCAT(s,
    APPLY(REPLICATE_1,s))
...
#define REPLICATE_10 \
  (THUNK)(BODY_10)(1)()
#define BODY_10(s) \
  (EXCEPTION) \
  (Too much recursion)
```

Higher-order functions

- Self-application: solved
- Nested application of an arbitrary function: needs further research

```
appFst :: (x -> y)
       -> (x,b) -> (y,b)
appFst f (a,b) = (f a, b)
```

```
appFst (appFst (+1))
```

```
APPLY(APPFST_0,
      APPLY(APPFST_1,PLUS_1))

#define APPFST_0 \
  (THUNK)(BODY_0)(2)()
#define BODY_0(f,a) // perform

#define APPFST_1 \
  (THUNK)(BODY_1)(2)()
#define BODY_1(f,a) // perform
```

Summary

- `hs2cpp`: Haskell $\rightarrow$ CPP - for building C macro libraries
- Full support:
  - Primitive values, large text, algebraic data types
  - Pattern matching, local definitions functions
  - Parametric polymorphism, type classes
  - Every language feature and extension that are desugared to these
- Partial support:
  - Recursion
  - Higher order functions
- `hs2cpp` GHC plugin
  https://github.com/nboldi/hs2cpp