Julia2C: a whole-program source-to-source translator

Hongbo Rong Todd A Anderson Hai Liu
Intel Programming and Systems Labs (PSL)
hongbo.rong,todd.a.anderson,hai.liu@intel.com

1. Introduction

Julia2C is a source-to-source translator from Julia to C. It converts basic Julia types and expressions into corresponding C types and statements.

By translating Julia to C, we leverage Julia's high-level abstractions (matrix, vector, ..), which are easier to analyze, and can potentially expose their massive data parallelism with the rich extensions of C (like OpenMP vectorization and parallel for pragmas). The tool may also extend Julia to new architectures where the only available tool chain is for C.

Julia is a new scripting language for technical computing [1], appealing in several aspects: the clean syntax, the high-level abstractions including matrix and vector and their operations, optional static typing, and its advanced LLVM compiler framework. Still in its early stage, Julia has attracted attention in the field of high-performance computing.

In developing a Problem-Solving Environment (PSE) based on Julia, we analyzed and exposed the data-parallelism in Julia operations so that they can be decomposed into many smaller operations and run in parallel in different threads. However, the multi-threading support in Julia is still under development. This specific implementation issue brought birth to Julia2C, although later it becomes clear that the significance of Julia2C is beyond that specific issue.

This paper briefly introduces the main idea and implementation of Julia2C. It has been open sourced, and can be freely accessed at https://github.com/IntelLabs/julia/tree/j2c.

2. Whole-program translation

We translate a Julia function, including all its callees, direct or indirect, to C. So the entire call graph, with that Julia function as the unique root, is translated. Thus it is a whole-program translation starting from a root function.

The C code is then compiled by a C compiler into a shared library. In the original Julia program, every call to the root function is replaced by a call to the corresponding C root function in this shared library.

The main reason why we pursue a whole-program translation approach is to make the C root function self-contained, independent of Julia runtime. We do not expect somewhere in the execution of the C root function, some Julia runtime functionality is required in order to proceed. There is a practical difficulty: for new architectures like Intel MIC, Julia cannot be built to generate correct code, as its internal compiler LLVM may be not ready. Also frequent boxing/unboxing data and switching execution between Julia and C would make debugging hard.

During the whole-program translation process, all callees are to be translated, conceptually. Some of the callees are Julia user functions, while the others are Julia runtime functions, which are written in Julia as well. All these Julia functions are translated in the same way. However, some Julia runtime functions would not be translated easily or necessarily: they manipulate complicated Julia internal data structures like dictionaries or code cache, which do not have to be exposed in C. Instead, for such a Julia runtime function, a C function with high-level equivalent semantics but without the low-level Julia internal implementation details is manually written. This is the only exception in the whole-translation process. The manually-written C functions for such Julia runtime function together are named as *runtime kernel* in this paper.

2.1 Data structures and translation

Fig. 1 shows the main data structures. Each Julia function's Abstract Syntax Tree (AST) is translated to a *result tree*. The two trees' nodes are 1-1 corresponding. In the result tree, each node points to its parent and children, and points to a prolog and an epilog string. The strings are C expressions or statements (for example, "x+y"), generated when the AST node is translated into this result node. A node also has a variable $result\_var$ to represent its value.

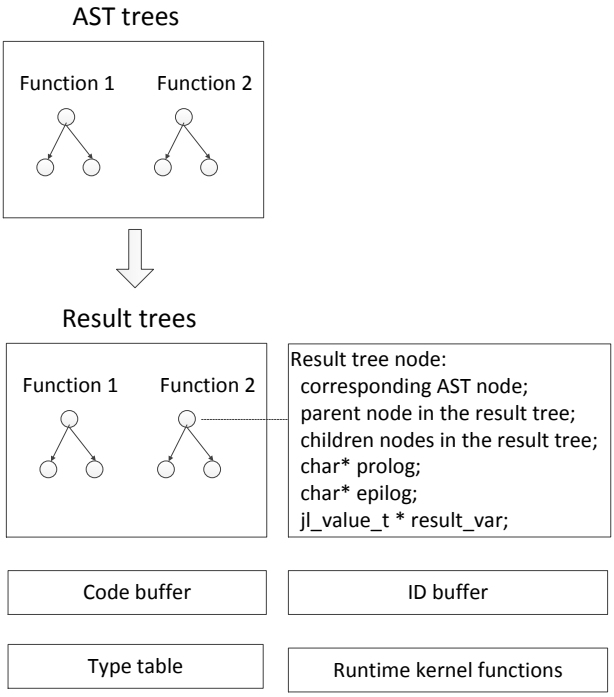

Figure 1: Main data structures

All the strings are stored in a code buffer, except the identifiers are stored in a separated ID buffer. Both buffers are sequential: they are written in one direction, and never reclaimed.

A type table contains all the C types built during the translation process. Each type is a C string (for example, "int"). Basic Julia types are translated straightforward to basic C types. Julia array type (e.g. Array{Float64, 1}) is converted into a pointer (e.g. "double *"). More complicated Julia types like tuple and struct are translated into C structs, with the fields translated recursively. A Julia type is compared with all the already-translated Julia types for equivalence, and is translated only when it has not been.

The translation process of a function is a post-order traversal of the AST tree. For each AST node, all its children are traversed first. Then the node is translated to a result tree node. According to the semantics of the AST node, the prolog and epilog string for the result tree node are generated, stored in the code buffer.

For example, consider a Julia AST node "+" with two children "x" and "y", which are two integers. The children have been translated before due to the post-order traversal, and thus their result tree nodes are already built and their fields are filled.

Now for the current AST node "+", a result tree node is built, and connected with the two children result tree nodes as parent-child relationship; a temporary symbol is generated as the *result_var*; at the same time, the type of the temporary symbol is checked, and if it has not been converted, it is converted into a C type and stored in the type table; and a C string "x+y" is generated as its epilog.

After the root function and all its callees have been translated, it is the time to dump all the result trees into a single C file. Mainly, all the C types created are dumped, followed by the declarations of all the functions, and then their definitions. The declarations are dumped first to handle recursive function calls, if any.

In dumping a function's definition, its result tree is traversed: For a result tree node, its prolog is dumped first, then all the children are traversed; after returning to this node, the epilog of the node is dumped.

2.2 Garbage collection and unboxing/boxing of data

Julia is a managed language, where an array object is freed of memory by a garbage collector if that object is found dead. In a Julia function, there can be explicit allocation of arrays, but no explicit deallocation of them. In translating it to a C function, we figure out the array to return to the caller of the root function, keep it and free all other (temporary) arrays.

Julia data are unboxed before passed to a C function as parameters. The C function may return a C array, which may be boxed in the caller Julia function.

3. Conclusion and future work

We have briefly introduced Julia2C, a source-to-source translator aiming to leverage Julia's high-level abstractions and C's parallel support and target new architectures. So far, it is combined into the LLVM IR code generation phase of Julia. A better way might be to factor it out as a separate module or external package, and thus make it usable in any phase of Julia.

References

[1] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," http://julialang.org/images/julia-dynamic-2012-tr.pdf.