Bull. Nov. Comp. Center, Comp. Science, 49 (2025), 37-48
@ 2025 NCC Publisher

Methods and tools for constructing specialized versions of universal Cloud Sisal programs

Victor N. Kasyanov, Elena V. Kasyanova

Abstract. The paper considers the problem of constructing the specialized versions of universal Cloud Sisal programs and presents an approach to solving it with the help of reducing concretizations. It describes a cross-platform Cloud Sisal compiler of the cloud parallel programming system (CPPS), which performs these transformations during compilation.

Keywords: compiler, program concretization, internal representation, reducing transformation, programming system, Cloud Sisal language, annotated programming

Introduction

The CPPS, developed at the Institute of Informatics Systems SB RAS (IIS SB RAS) and available for use via a web browser, is aimed at providing the user with tools for the development, debugging, verification and execution of parallel programs according to their functional specifications in the Cloud Sisal language [1 - 8]. The Cloud Sisal language continues the traditions of the previous versions of the SISAL language, remaining a functional streaming language oriented towards writing large scientific programs. At the same time, it expands their capabilities with cloud computing support [9 - 12].

By program concretization, or program optimization in a given context, we mean such a transformation of a program within the framework of one language that preserves the meaning of the program and improves its quality in a predefined subset of its applications relative to a certain quality criterion [13, 14]. The subclass of so-called reducing concretizations (or reductions) has the following characteristic feature: an improvement of the quality of a given universal (or general-purpose) program with the help of reductions is not due to its restructuring, but to removing the objects and constructions that become redundant after the set of tasks to be solved is narrowed in the given restricted context [15].

The paper presents a solution of the problem of constructing the optimized specialized versions of universal Cloud Sisal programs in the CPPS with the help of reducing concretizations of annotated programs.

The paper is organized as follows. More information about the CPPS and the Cloud Sisal language is given in Section 1. The internal graph representation of Cloud Sisal programs used by the CPPS is considered in Section 2. Section 3 proposes a set of reducing concretizations for universal Cloud Sisal programs and discusses its properties. Section 4 describes a cross-platform Cloud Sisal compiler of the CPPS which carries out these transformations during compilation. Finally, a conclusion is given in Section 5.

1. The CPPS and Cloud Sisal Language

Modern approaches to parallel program development are mainly architecture-oriented: to achieve efficient operation, the programs being constructed are closely linked to the architectures of parallel computing systems on which they are executed and, as a rule, developed. Therefore, the requirements for the qualifications of parallel program developers are exceptionally high, especially since parallel programs are much more difficult to test and debug than sequential programs, and the problem of parallel programs verification is far from both practical and theoretical solution. Moreover, only a narrow circle of domestic users has access to high-performance computing equipment, which in terms of the number of supercomputers and their total capacity is much lower than in developed countries. In addition, the high-performance equipment is based in a limited number of locations, while the bulk of application programmers work in other environments, where parallel programs cannot be developed. Therefore, the project [1- 9] being carried out at the IIS SB RAS to develop the Cloud Sisal language and the CPPS supporting the development, verification and debugging of the architecturally independent parallel Cloud Sisal programs and their correct transformation into an efficient code for the parallel computing systems of various architectures using semantic transformations in clouds seems quite promising.

The Cloud Sisal language features the advantages typical of functional programming languages, such as single assignment and deterministic results for parallel and sequential implementation, but it includes arrays and loops and uses the semantics of always-terminating computations [9]. The Cloud Sisal language also supports annotated programming and program concretization [13 - 18]. It allows a user to describe the known semantic properties of a program in the form of formalized comments called annotations (or pragmas). In particular, with the help of these pragmas, the user can describe a narrowed context of application of a universal Cloud Sisal program.

For example, the pragma "assert = Boolean condition" can be located in the header of a function definition before the first formal parameter and can set conditions on the parameter values specified in it, which must be true for any function call immediately before the execution of the body of this function. In particular, the Boolean condition of the pragma can have a simple kind “parameter = constant” and describe the known constant value of the input parameter of a general-purpose function in the described narrowed context of its application. These known constant values of the input parameters can be used for the specialization of the function via mixed computations [19 - 21] or partial evaluation [22]. It should be noted, however, that the assert pragmas can be used to describe more general and complicated restrictions on the values of input parameters of general-purpose functions. For example, the assert pragma in the header of the function “foo” specifies that the function is used in the context where its first input parameter is less than ten

```
function foo(//$ assert=n<10 integer n

              real x returns real)

         if n<10 then foo1(n+10,x*10)

         elseif n<20 then foo2(n+20,x*20)
```

```
           else foo3(n+30, x*30)

           end if

  end function
```

and therefore the function can be replaced by the following specialized version

```
  function foo(//$ assert=n<10 integer n

              real x returns real)

           foo1(n+10,x*10)

  end function
```

Another example is the "non_used = list of values" pragma, which allows us to describe a stable context of a general-purpose program by narrowing the set of its possible results used in this context. This pragma, placed before the "returns" keyword in the function header, contains a list of values which are not used in any function call, and, therefore, the calculation of these values can be removed from the function body. For example, as indicated in the annotation below, the second result array of the “gen” function is never used in the context described here:

```
  function gen( N : integer

               //$ non_used=_[2] returns TwoDim, TwoDim)

      for i in 1, N cross j in 1, N do

      returns array of i * j; array of i + j

      end for

  end function
```

and, therefore, its calculation in the function body can be deleted:

```
  function gen( N : integer

               //$ non_used=_[2] returns TwoDim, TwoDim)

      for i in 1, N cross j in 1, N do

      returns array of i * j

      end for

  end function
```

An annotated Cloud Sisal program can be considered as a program written in the Cloud Sisal language extended by annotations (or pragmas) which are formalized comments relevant to the semantics of a basic program to be annotated [13-18, 23-25].

The extensions of high-level languages by special annotations (pragmas) commonly used in compilers are currently considered to be part of language description [26 - 28]. One of the three main approaches to transformational program development is the so-called extended compilation [29] characterized by advice permission and partial relaxation of restrictions on the basic language. That is, the transformation system accepts not only the basic program, but also some annotations as guidance on transformations. Importantly, annotations of the Cloud Sisal programs are more than directives or hints for a compiler or another transformation system (e.g., for automatic parallelization of a sequential program): they can be used to modify the semantics of the basic program, though only moderately.

It is assumed that a general-purpose program can be annotated by the information known about a specific context of its applications. Through this, annotations added to the basic general-purpose program specify a covering context. It means that any actual program application from the context described should be admissible for annotations, though some admissible applications may be beyond the context.

Annotations may specify the context of a basic program application both explicitly, as assertions which are predicate constraints on the admissible properties of program fragments or admissible states of computations, and implicitly, as directives specifying the admissible transformations of annotated programs or states of computations in the indicated points of a program.

A concretizing transformation of a general-purpose program annotated by the information about a specific context of its applications is a replacement of the annotated program with its specialized version equivalent to (or correct with respect to) the original one on the context-defined ranges of inputs and outputs and is better than the source program by the quality criteria given by the context. Thus, the annotated program is subjected to concretizing transformations as a whole. It means that the concretizing transformations can change not only the basic program but its annotations as well.

It was shown [13 - 18, 23 - 25] that the class of correct transformations of annotated programs covers various kinds of manipulations with basic programs.

In particular, the approach allows specializing and generalizing the transformations of basic programs to be reduced to the equivalent transformations of annotated programs.

Another advantage of the approach is the possibility to perform global transformations of basic programs by the iterative application of elementary (context-free) transformations of annotated programs.

It is proved [25] that any basic program transformation represented by a normal Markov algorithm [30] may be modeled within the annotated program framework in such a way that annotations specify only the elementary transformations of annotated programs and for any basic program the transformation process not only has the same result as the modeled normal algorithm, but also performs a similar sequence of processing steps.

2. Internal representation of Cloud Sisal programs

The CPPS uses an internal representation of Cloud Sisal programs oriented towards their semantic and visual processing and based on attributed hierarchical graphs with ports [31, 32].

Within the framework of this representation, the program is assembled from modules before its interpretation or optimizing compilation. The following essential requirements were taken into account when developing the internal representation: machine independence, completeness, the possibility of retranslation, simplicity of interpretation, structuring of objects, explicit representation of all implicit actions on data, and extensibility [1, 7].

The nodes of the internal representation graph correspond to the expressions of a Cloud Sisal program, and the arcs reflect the data transfers between the node ports, the ordered sets of which are assigned to the nodes as their arguments (input ports, or inputs) and results (output ports, or outputs). So, the nodes of the graph describe the actions on their inputs (arguments), the results of which are received at the outputs of the nodes and are sent along arcs to the inputs of other nodes.

Due to the property of the Cloud Sisal language, the internal representation graph is acyclic and does not contain two arcs entering the same input.

The nodes of the internal representation graph can be simple or composite.

Simple nodes (or simply nodes) correspond to elementary expressions. They have no internal structure and represent elementary operators, such as plus or minus, under their arguments. There is a special kind of simple nodes, each of which has one output and an empty set of inputs — they represent literals (or constants).

Composite nodes (or fragments) correspond to composite expressions of a Cloud Sisal program, such as a loop expression or a function body. Each fragment is a subgraph composed of a set of subfragments corresponding to subexpressions of which the composite expression consists. Any of these subfragments can be a simple node.

It is assumed that the nesting tree of fragments is ordered, i.e. a set of sons of any node of the tree is linearly ordered. Moreover, this ordering of nodes does not contradict the direction of the arcs connecting the ports of these nodes.

Thus, the internal representation graph of Cloud Sisal programs, unlike the control flow graph [13, 33] typically used in optimizing compilers for imperative languages (such as C or Fortran), expresses not the control flow but the data flow in the program. In addition, it also preserves the existing hierarchy of language constructs in the translated program, typically expressed in compilers within an intermediate representation graph of the translated program such as a derivation tree [13, 33].

The internal representation graph of Cloud Sisal programs also has a number of useful properties including the following two [1, 7]:

1. Explicitly defined information (semantic) connections (arcs) between operands of expressions (ports of nodes) make it possible to interpret a Cloud Sisal program without additional transformations. This entails the absence of side effects of calculations (due to the absence of the concept of a variable) — a natural property of purely functional languages.

2. At the level of individual information-independent operations, parallelism is clearly represented, which does not depend on the machine architecture.

3. Reducing concretizations of Cloud Sisal programs

A set of reducing concretizations for general-purpose Cloud Sisal programs has been developed, covering all the main language constructs and all the main ways of their specialization [34]. Most of these are context-free transformations of Cloud Sisal language expressions, which significantly simplifies not only the proofs of their correctness and feasibility of their application, but also their practical implementation.

Almost all the transformations from the developed set are equivalent. Exceptions are transformations of constant calculations for real or complex values, which may be nonequivalent (and possibly incorrect) due to a possible change in the calculation accuracy deriving from the fact that the compiler is launched on a computer different from the target one. Also, nonequivalent though correct are some analytical transformations, such as replacing by zero an expression which has the form of a multiplication of some subexpression by zero, due to a possible erroneous value of this subexpression.

It is assumed that a user can control the application of nonequivalent transformations using special pragmas. For example, a user can enable the application of a predefining analytical transformation to some expression either explicitly, by requiring the unconditional (context-free) application of the transformation to the part of the program where this expression is located, or implicitly, by placing a pragma about the absence of error values so that this expression is in the scope of this pragma.

Conventionally, the set of reductions is divided into the following subsets.

1. Substituting a constant value for a variable. Cloud Sisal is a single assignment language, so if a variable in a program has been assigned a constant value, or if a pragma specifies that the variable has a constant value, then the specified constant value can be substituted for the variable in all places where it is used. This transformation is not context-free, since it uses a set (e.g., in the form of a hash table) of variable-constant pairs found during program processing.

2. A set of reductions of simple expressions, including constant calculations and analytical transformations, which consists of 25 transformations.

3. A set of conditional expression reductions comprising 13 transformations. Cloud Sisal allows two types of conditional constructs: “if” and “case”. A natural reduction of a conditional expression is the removal of all branches with identically false or impassable conditions. For if-expressions, the latter means that if there is a branch with an identically true condition, then all branches following it can be removed. If after the reduction of an if-expression only the “then” or “else” branch is left, it can be replaced by a list of expressions of the results of this branch. If the “then” branch is removed, but at least one “elseif” branch is preserved, the Boolean condition and the “then” subbranch of this branch can be substituted for the Boolean condition and the “then” branch of the expression being processed. If after the reduction the if-expression has no branches left, it can be replaced by a list of error values of the same dimension as the results returned by the branches of the original construct. For a case-expression, if all test values are identically false, then “case” can be replaced by the contents of the “else” branch or, in its absence, by a list of error values of the appropriate dimension.

4. The set of reductions of let-expressions consists of the following transformations. The first removes the definitions of the local variables having constant values, by replacing occurrences of these variables with their values. The second transformation is similar to the first one, but it removes a local variable and replaces it with its corresponding expression if it is used only once. The third transformation is the removal

of definitions of local variables not used within their scope. Finally, if a let-expression does not contain variable definitions or the list of its result expressions does not depend on its local variables, then this expression can be reduced to this list.

5. Reductions of functions and their calls. The body of a (non-recursive) function can be substituted in the place of its single call as a let-expression. If program analysis has revealed or a pragma has specified that specific results of some function are not used after any call, the function can be rebuilt so that it does not return these results, and the expressions that calculate them can be removed from it. A similar approach can be taken if, in all calls to a function, a certain argument is equal to the same known constant. In such a situation, the function can be rebuilt so that it does not accept this argument, and the known constant value can be substituted in the function body in any place where it is used.

6. Loop reductions. Loop unrolling is generally not a reducing transformation as it may increase the code size and change its structure. Also, since test-driven loops in Cloud Sisal can be asynchronously parallel, unrolling them could negatively affect parallelism. However, in the case when the number of loop iterations is 0 or 1, the loop can be reduced to its body and/or to the expressions computing the loop reductions. Also, when the loop body is empty or does not depend on previous iterations and counters, the loop can be reduced to the known expressions computing its reductions.

7. Removing unused computations. If the value of some expression is not used further in the program or is excluded from the list of its results by the corresponding pragma, this expression can be deleted or replaced by an error value. This transformation is not context-free.

The properties of this set of reducing concretizations have been studied from the point of view of the mutual influence of transformations (repetition and deadlock relations between transformations) and the influence of the order of application of transformations on the result obtained. In particular, it has been shown that the set does not have the Church-Rosser property [33].

An effective three-pass strategy for applying the transformations from the developed set of reductions has been formulated and substantiated.

4. Cross-platform Cloud Sisal compiler

In addition to the optimizing compiler from the CPPS using Windows and building the C# code, a cross-platform compiler CS2CPP for the Cloud Sisal language has been developed and implemented in Python [34].

The CS2CPP compiler translates Cloud Sisal programs into C++ programs extended by OpenMP directives [35]. It is divided into three parts (parser, reduction block and code generator), which interact with each other by transmitting the internal representation graph of a Cloud Sisal program in the form of the JSON text [36].

The parser builds the internal representation graph of a source Cloud Sisal program, which can be passed, at the user's option, to another part of the compiler (reduction block or generator) either directly or indirectly, following a preliminary processing of this representation using other components of the CPPS, such as a visual debugger.

The reduction block transforms a Cloud Sisal program within its internal representation and builds its specialized version using our developments: the set of reducing concretizations and strategy for their application.

The code generator, based on the received internal representation of a Cloud Sisal program, uses mixed code generation methods to construct its optimized C++ code extended by OpenMP directives. In particular, it implements memoization methods to optimize the generated C++ functions. The code generator uses abstractions for Cloud Sisal data types which are necessary for implementing error values. Error values are set transparently for the main program code and do not require additional code in the output program in the C++ language.

All the three parts of the cross-platform CS2CPP compiler are included in the CPPS environment, together with the support for execution mode. In this mode, the C++ code obtained by the compiler is processed using the free optimizing compiler GCC [37]. The main function of the Cloud Sisal program is renamed to the sisal_main function, and the main function of the C++ program loads the input data and calls sisal_main using the input data as arguments. The names of the remaining functions in the C++ program correspond to the names in the source program in the Cloud Sisal language. The resulting executable files accept the program input data (the arguments of the main function) via stdin in the form of a JSON text (the JsonCpp library [38] is used), which facilitates working with the compiler in the automatic mode and using it in other systems, such as development environments. The presence of the required arguments of the main function among the input data is also checked by the program. The calculation results are also put out to standard output as a JSON text.

A system for automatic compiler testing has been developed with a set of some test Cloud Sisal programs and sets of their input data and expected correct calculation results. For each test Cloud Sisal program, several sets of input and expected output data can be specified.

The proposed compiler architecture allowed the development of its three parts independently of each other and made it simple to use them in the CPPS, both separately and together with other CPPS components. In particular, it provided the ability to extend the CS2CPP compiler to other output languages by writing new code generators.

The LLVM [39] is a widely used software infrastructure for creating compilers and software development tools. It includes a range of frontends for many high-level languages, a modern source- and target-independent optimizer and code generation support for many popular CPUs (as well as some less common ones!). At the core of the LLVM is an intermediate representation of the code (LLVM IR code), which can be transformed during compilation, linking, and execution. From this representation, the optimized machine code is generated for a range of platforms, both statically and dynamically.

We have started work on creating a new code generator that constructs LLVM IR code of a Cloud Sisal program based on its internal representation. The code generator is being implemented in Python using the LLVM Lite library [40].

5. Conclusions

The paper presents an approach to solving the problem of constructing specialized versions of universal Cloud Sisal programs in the cloud parallel programming system (CPPS) with the help of reducing concretizations. The cross-platform CS2CPP compiler of the cloud system is described. It implements this approach in the process of translating the Cloud Sisal language into the C++ language extended by OpenMP directives. Our current plans are to expand the CS2CPP compiler to integrate the CPPS with the LLVM infrastructure. This will allow us, on the one hand, to use existing tools and libraries of the LLVM infrastructure within the CPPS, and on the other hand, to provide support for the Cloud Sisal language by the LLVM infrastructure.

Acknowledgments. The authors are grateful to all colleagues who took part in the work discussed in the paper.

References

[1] Kasyanov V.N., Kasyanova E.V. Methods and system of cloud parallel programming // Problems of optimization of complex systems (Part 1). Proc. XIV International Asian School-Seminar. – Almaty. – 2018. – P. 298-307 (In Russian).

[2] Kasyanov V., Kasyanova E. Methods and system for cloud parallel programming // 21st International Conference on Enterprise Information Systems. Proc. ICEIS 2019. – 2019. – Vol. 1. – P. 623–629.

[3] Kasyanov V.N., Kasyanova E.V. Methods and tools for formal verification of Cloud Sisal programs // International Conference on Mathematics and Computers in Science and Engineering. Proc. MACISE. – 2020. – P. 219-222.

[4] Kasyanov V.N., Kasyanova E.V., Kondratyev D.A. Formal verification of Cloud Sisal programs // Journal of Physics: Conference Series. – 2020. – Vol. 1603. – P. 012020.

[5] Kasyanov V.N., Kasyanova E.V., Zolotuhin T.A. Visualization of data-flow programs // Lecture Notes in Electrical Engineering. – 2019. – Vol. 574. – P. 119-124.

[6] Kasyanov V.N., Zolotuhin T.A., Gordeev D.S. Visualization methods and algorithms for graph representation of functional programs // Programming and Computer Software. – 2019. – Vol. 45. – No. 4. – P. 156–162.

[7] Kasyanov V.N., Zolotukhin T.A., Gordeev D.S. et al. Cloud Parallel Programming System CPPS: Visualization and Verification of Cloud Sisal Programs. – Novosibirsk: IPC NSU, 2020 (In Russian).

[8] Kondratyev D.A., Promsky A.V. Towards verification of scientific and engineering programs. The CPPS project // Computation Technologies. – 2020. – Vol. 25, No. 5. – P. 91-106.

[9] Kasyanov, V. N., Kasyanova, E. V. Programming Language Cloud Sisal. – Novosibirsk, 2018. –(Preprint / Institute of Informatics Systems of the Siberian Branch of the Russian Academy of Sciences, No. 181) (In Russian).

[10] Feo J.T., Cann D.C., Oldehoeft R.R. A report on the Sisal language project // Journal of Parallel and Distributed Computing. – 1990. – Vol. 10, No. 4. – P. 349-366.

[11] Gaudiot J.-L., DeBoni T., Feo, J., et al. The Sisal project: real world functional programming // Lecture Notices in Computer Science. – 2013. – Vol. 1808. – P. 84–72.

[12] Kasyanov V.N. Sisal 3.2: functional language for scientific parallel programming // Enterprise Information Systems. – 2013. – Vol. 7, No. 2. – P. 227–236.

[13] Kasyanov V.N. Optimizing transformations of programs. – M: Nauka, 1988 (In Russian).

[14] Kasyanov V.N. Program annotation and transformation // Programming and Computer Software. – 1989. – Vol. 15, No. 4. – P. 155-164.

[15] Kasyanov V. N. Reducing transformations of programs // Translation and optimization of programs. – Novosibirsk: Computing Center of the Siberian Branch of the USSR Academy of Sciences, 1983. – P. 86-98 (In Russian).

[16] Kasyanov V.N. Transformational approach to program concretization // Theoretical Computer Science. –1991. – Vol. 90, No. 1. – P. 37-46.

[17] Kasyanov V.N. A support tool for annotated program manipulation // Fifth European Conf. on Software Maintenance and Reengineering. Proc. – 2001. – P. 85-94.

[18] Kasyanov V.N., Mirzuitova I.L. Slicing: Program Slices and their Applications. – Novosibirsk: IIS SB RAS, 2002 (In Russian).

[19] Ershov A.P. Organization of mixed computations for recursive programs // Doklady Akademii Nauk SSSR. – 1979. – Vol. 245, No. 5 – P. 1041-1044 (In Russian).

[20] Ershov A.P. Mixed computation in the class of recursive program schemata // Acta Cybernetica. –1978. – Vol. 4, No. 1. – P. 19-23.

[21] Ershov A.P. Mixed computation: potential applications and problems for study // Theoretical Computer Science. –1982. – Vol. 18, No. 1. – P. 41-67.

[22] Jones N.D. An introduction to partial evaluation // ACM Computing Surveys. – 1996. – Vol. 28, Issue 3. – P. 480-503.

[23] Kasyanov V.N. Practical approach to program optimization. – Novosibirsk, 1978. – (Preprint / Computer Center of Siberian Division of the USSR Academy of Science, No.135) (In Russian).

[24] Kasyanov V. N. Annotated program transformations // Lecture Notes in Computer Science. – 1989. – Vol. 405. – P.71-180.

[25] Kasyanov V.N. On completeness of mechanism of annotation-directives // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 1995. – Iss. 3. – P. 59-68.

[26] Reference Manual tor the Ada Programming Language. – Springer-Verlag, 1983.

[27] Koelbel C.H., Loveman D.B., Schreiber R.S. et al. The High Performance Fortran Handbook. – The MIT Press, 1993.

[28] Lohr K.-P. Concurrency annotations for reusable software // Comm. of the ACM. – 1993. – Vol. 36, No. 9. – P. 81-89.

[29] Feather M.S. A survey and classification of some program transformation approaches and techniques // Program Specification and Transformation. – 1987. – P. 165-195.

[30] Markov A.A., Nagornij N.M. Theory of Algorithms. – Moscow: Nauka, 1984 (In Russian).

[31] Kasyanov V.N. Methods and tools for information visualization on the basis of attributed hierarchical graphs with ports // Siberian Aerospace Journal. – 2023. – Vol. 24, No. 1. – P. 8–17.

[32] Kasyanov V.N. Kasyanova E.V. Information visualization based on graph models // Enterprise Information Systems. – 2013. – Vol. 7, No. 2. – P. 187–197.

[33] Kasyanov V.N., Evstigneev V.A. Graphs in Programming: Processing, Visualization and Application. – St. Petersburg: BHV-Petersburg, 2003 (In Russian).

[34] Kasyanov V.N., Kasyanova E.V., Malyshev A.A. Cross-platform Cloud Sisal compiler for the cloud parallel programming system CPPS // Marchukovsky Scientific Readings 2024: Abstracts of the International Conf. Proc. – Novosibirsk: ICM&MG SB RAS. – 2024. – P. 135-135.

[35] OpenMP – URL: http://www.openmp.org.

[36] JSON – URL: https://www.file-extension.info/format/json.

[37] GCC – URL: https://gcc.gnu.org/.

[38] JsonCpp – URL: https://sourceforge.net/projects/jsoncpp.mirror/.

[39] LLVM Compiler Infrastructure – URL: https://llvm.org.

[40] LLVM Lite – URL: https://github.com/numba/llvmlite.

[41] 36. JSON – URL: https://www.file-extension.info/format/json

[42] 37. GCC – URL: https://gcc.gnu.org/

[43] 38. JsonCpp – URL: https://sourceforge.net/projects/jsoncpp.mirror/

[44] 39. LLVM Compiler Infrastructure – URL: https://llvm.org

[45] LLVM Lite Library – URL: https://github.com/numba/llvmlite