International Journal of Software Engineering
and Knowledge Engineering
Vol. 32, No. 7 (2022) 1043–1069
© World Scientific Publishing Company
DOI: 10.1142/S0218194022500401

Generation of C++ Code from Isabelle/HOL Specification

Jiang Dongchen* and Xu Bo†

School of Information Science and Technology
Beijing Forestry University, Beijing 100083, P. R. China
**jiangdongchen@bjfu.edu.cn*
†mu001999@bjfu.edu.cn

Received 22 February 2022
Revised 21 June 2022
Accepted 21 June 2022
Published 5 August 2022

Automatic code generation plays an important role in ensuring the reliability and correctness of software programs. Reliable programs can be obtained automatically from verified program specifications by code generators. The target languages of the existing code generators are mainly functional languages, which are relatively less used than C/C++. As C/C++ is widely used in the industry and many fundamental software facilities and the correctness verification of C/C++ programs is difficult and cumbersome, this paper provides an automatic conversion framework that allows to generate C++ implementation from verified Isabelle/HOL specifications. The framework is characterized by combining the verification convenience of Isabelle/HOL and the efficiency of C++. Since the correctness of the functional Isabelle/HOL specification can be guaranteed by interactive proofs, the correctness of the relevant generated C++ implementation can also be maintained.

Keywords: Code generation; C++; Isabelle/HOL.

1. Introduction

Isabelle/HOL is a general proof assistant that allows its users to express mathematical formulas and proofs in high-order logic [1]. It builds a formal environment for interactive theorem proving, where many tools and libraries are included for interactive verification. Besides mathematical theorems, some algorithms and large-scale applications can also be formalized in Isabelle/HOL [2], where their correctness is guaranteed by formal verification.

Once a specification is verified, it is often desirable to derive executable codes from the specification. Generally, the codes are manually written by experienced programmers or automatically generated by a code generator. While the manual reimplementation might incur some errors and can be cumbersome especially when

*Corresponding author.

efficiency is considered, automatic code generation is applied and applauded for reliability consideration. Isabelle/HOL allows converting executable specifications into codes in SML, OCaml, Haskell and Scala [3]. But these languages of the generated codes are functional, which are not commonly used in industrial development. Comparatively, C/C++ is more widely used in mainstream fundamental software facilities development, e.g. operating systems and compilers. As a result, most research teams tend to formalize these typical software in Isabelle/HOL or Coq, and interactively verify the correctness of their specifications.

Aiming to combine the verification convenience of Isabelle/HOL and the efficiency of C++ language, this paper proposes a framework named Isabelle2Cpp to generate executable C++ codes from Isabelle/HOL specifications. With it, the ready-made Isabelle/HOL specifications can be reused in industrial development and the correctness of codes can be guaranteed by the existing proofs. Programmers only need to write and verify formal specifications in Isabelle/HOL, while their corresponding C++ codes can be correctly generated by Isabelle2Cpp.

This paper is organized as follows. Section 2 reviews the related works. Section 3 describes the general framework Isabelle2Cpp. Section 4 proposes the definition-based conversion from Isabelle/HOL specifications to C++ codes, which includes both the type conversion and function conversion. Section 5 proposes the rule-based conversion. Examples to assess the usability of the conversion framework are presented in Section 6: the merge sort algorithm and the Isabelle/HOL library file List. thy are used. Section 7 concludes the whole paper.

2. Related Work

In general, there are two ways to guarantee the correctness of a program by formal methods: one is to obtain executable codes from formally verified specifications by some code generators; the other is to formalize the existing programs in a theorem prover (e.g. Isabelle/HOL), and then to prove the correctness of the formal description.

Isabelle/HOL supports code generation for many functional programming languages (SML, OCaml, Haskell and Scala), which allows the conversion from Isabelle/HOL specifications to functional programs in these target languages [3]. The code generator of Isabelle/HOL provides standard target entities in target languages for abstract entities in Isabelle/HOL and guarantees the partial correctness of the generated programs. Based on the code generator, Florian Haftmann *et al.* further propose the Isabelle Refinement Framework (IRF) to support data refinement. The IRF features a stepwise refinement approach and enables specifying concrete data structures for abstract ones by including relevant equation theorems. By IRF, algorithms on abstract datatypes, e.g. set, can be refined to a concrete efficient implementation with specified concrete data structures [4]. As Isabelle/HOL functions produce no side-effect, IRF are used to generate purely functional programs.

Lukas Bulwahn et al. propose the Imperative/HOL to generate imperative functional programs in Haskell and ML [5]. The new framework includes a new heap abstraction for program formalization and a set of rules for imperative specification verification in Isabelle/HOL. Peter Lammich proposes a separation logic framework which enables refining abstract specifications into imperative implementations in Imperative/HOL; meanwhile, an imperative collection framework is also proposed to support standard imperative data structures for the existing abstract datatypes [7]. With this framework, imperative codes of OCaml, SML, Haskell and Scala can be generated from the imperative specifications in Imperative/HOL. Peter Lammich also proposes an imperative language Isabelle-LLVM embedded in Isabelle/HOL, which allows to generate low-level LLVM IR from abstract Isabelle/HOL specifications in IRF [8].

In the area of imperative program verification, it is necessary to formalize the imperative programs first. Different techniques and tools are used to bridge the gap between low-level implementations and higher-level abstract representations. For example, Farhad Mehta and Tobias Nipkow develop a sound modeling and reasoning method for imperative programs with pointers in high-level logic [9]; Greenaway *et al.* propose the tool AutoCorres that can automatically abstract low-level C semantics into a high-level representation in Isabelle/HOL [10]. Besides those general tools and methods, some fundamental software, such as operating systems (OS) and compilers, are also formalized and their correctness are verified. Klein *et al.* present seL4 [11], a high-performance OS microkernel, with its C implementation verified formally in Isabelle/HOL, which is the first formal proof of a complete OS kernel [12]. Leroy *et al.* investigate the formal verification of compilers, and CompCert [13] they present is the first commercially available optimizing compiler formally verified to be exempted from mis-compilation [14]. Besides the above-mentioned works, Krebbers and Wiedijk present the CH2O Project [15], which aims at formalizing the ISO C11 standard of the C programming language and allows writing formal specifications with formalized C semantics directly in a theorem prover. These works are for various applications and have seminal significance; many technical problems about operational semantics of the implementation languages in verification have been solved.

The conversion framework Isabelle2Cpp allows the automatic generation of C++ codes from Isabelle/HOL specifications. Its basic principles are similar to those of the existing code generation for functional programming languages in Isabelle/HOL. The main difference is that the target language of this framework is C++, which is a widely used non-functional imperative language. This difference leads to a big difference in implementation. Compared with the imperative program verification, Isabelle2Cpp generates executable C++ codes from the existing Isabelle/HOL specifications automatically, which allows users to focus on writing and verifying the specifications in Isabelle/HOL, while putting imperative C++ code generation work aside.

3. Isabelle2Cpp Framework

Theoretically, the operational semantics of the functional Isabelle/HOL specifications and the imperative C++ programs are equivalent. The difference lies in how types (datatypes) and functions are defined. To ensure the correctness of conversion from Isabelle/HOL specifications to the relevant generated C++ codes, two properties need to be satisfied: first, the overall structures of the specification and its corresponding C++ code need to be isomorphic; second, the operation semantics of their fundamental operations need to be equivalent.

To obtain the isomorphic structures, a standard intermediate representation (IR) is used to represent the structures of an Isabelle/HOL specification and its C++ codes. In the framework Isabelle2Cpp, abstract syntax trees (AST), which describe the syntax structures of datatypes and functions in Isabelle/HOL, are used as the standard IR: the Isabelle/HOL specification will be first converted into an AST, and the corresponding C++ codes will be generated from the AST. The basic type constructions and fundamental operations, such as value constructions, control-flows and basic data operations, can be constructed according to definitions. The architecture of Isabelle2Cpp is shown in Fig. 1, the conversion from AST to C++ code does all conceptional conversions, and the synthesis part produces concrete C++ files but does not change the program essentially.

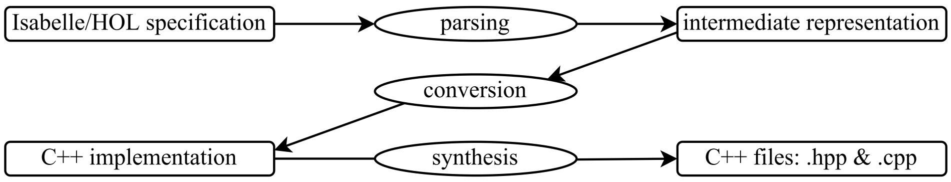

Fig. 1. Architecture of Isabelle2Cpp.

A theory of Isabelle/HOL consists of three parts: definitions of datatypes, definitions of functions and theorems with the corresponding proofs. Among them, datatype definitions and function definitions are related to the code execution, while theorems and their proofs guarantee the correctness of relevant functions. Therefore, the conversion from Isabelle/HOL to C++ mainly includes two parts: the conversion from Isabelle/HOL datatypes to C++ types and the conversion from the Isabelle/HOL functions to their semantically equivalent C++ implementation.

In Isabelle/HOL, datatypes are introduced by specifying type names, type variables and construction rules which include constructors and argument types. Among them, type variables are optional and are included when polymorphic types are defined. In C++, the structure similar to datatype in Isabelle/HOL is class (class template), and it is set as the target structure of datatype for conversion: The type name, type variables and construction rules of a datatype can correspond to the class

(class template) name, template parameters and nested classes in C++; the constructors in construction rules are corresponding to nested classes names, and the argument types are corresponding to the types of data members in the nested classes. The isomorphic structures and one-to-one correspondence between basic components guarantee the equivalence between the datatypes in Isabelle/HOL and the corresponding classes (class templates) in C++.

A function of Isabelle/HOL specifies the relationship between input and output, and its calculation is defined by pattern matching: if a pattern is matched, the relevant expression will be executed. Similarly, a function (function template) in C++ is set as the target structure for a function in Isabelle/HOL. The name, datatype and operations of a function in Isabelle/HOL can correspond to the name, type and compound statements of a function (function template) in C++. The order of operations in the generated C++ function are the same with the order of ones in the Isabelle/HOL function. Compound if-statements are used to generate pattern matching: if-conditions are constructed according to the constructors of relevant parameters; specific operations in expressions, such as value construction, recursive call, application of existing functions, If-expression, Case-expression, Let-expression and Lambda-expression, can be converted into compound control-flow statements composed of expression statements, declaration statements and selection statements.

For the implementation, a Isabelle/HOL specification with definitions of datatypes and functions will be parsed to the standard ASTs. The ASTs not only include the names, type variables and construction rules of relevant datatypes, but also include the names, prototypes and equations of relevant functions. Then the corresponding C++ classes (class templates) and functions (function templates) will be generated from the AST with applying equivalent mapping rules. After all, the headers of C++ standard library and the C++ files of corresponding head theories of the current specification are added. The conversion process is shown in Fig. 2.

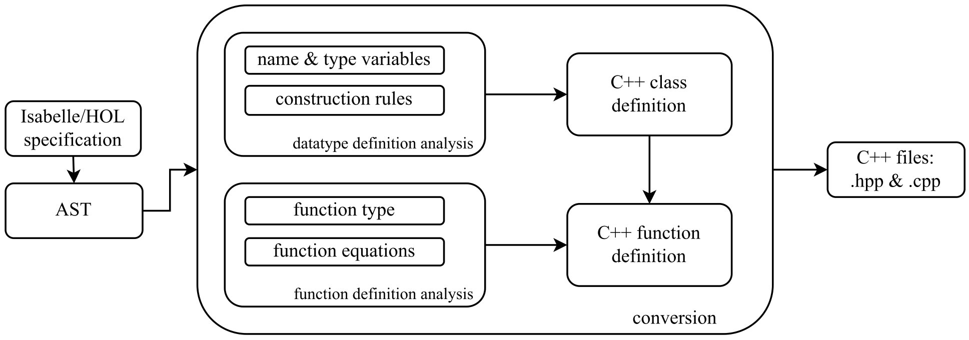

Fig. 2. Conversion process of Isabelle2Cpp.

4. Definition-Based Conversion

4.1. *Type conversion*

The syntax of Isabelle/HOL specifications follows the way of functional programming: both values and functions are represented by symbols. A datatype in Isabelle/HOL is a collection of values. Its definition emphasizes the way how a value is represented by the corresponding symbols. For a datatype t in Isabelle/HOL [16], the general form of its definition is

$$\begin{aligned}\textbf{datatype}\ ('a_1, \ldots, 'a_{\mathrm{n}})t = {} & \mathrm{C}_1 {''}\tau_{1,1}{''} \ldots {''}\tau_{1,\mathrm{n}_1}{''} \\ & \mid \ldots \\ & \mid \mathrm{C}_{\mathrm{k}} {''}\tau_{\mathrm{k},1}{''} \ldots {''}\tau_{\mathrm{k},\mathrm{n}_{\mathrm{k}}}{''}\end{aligned}$$

Here, $'a_1, \ldots, 'a_{\mathrm{n}}$ are type variables; $\mathrm{C}_1 {''}\tau_{1,1}{''} \ldots {''}\tau_{1,\mathrm{n}_1}{''}, \ldots, \mathrm{C}_{\mathrm{k}} {''}\tau_{\mathrm{k},1}{''} \ldots {''}\tau_{\mathrm{k},\mathrm{n}_{\mathrm{k}}}{''}$ are construction rules, where C_i is the ith constructor and $\tau_{i,1} \ldots \tau_{i,\mathrm{n}_i}$ are its argument types. Each construction rule provides an independent way to construct a value of the datatype.

A datatype is either recursive or non-recursive: it is recursive if some of its construction rules contain the datatype itself directly or indirectly; otherwise, it is non-recursive and all its construction rules only contain existing types. It should be noted that type variables and construction parameters are optional: a datatype without type variables and construction parameters is the simplest enumeration type, while a datatype with type variables is a polymorphic type. For example, **datatype** *bool* $=$ *True* | *False* is the enumeration type of Boolean values, **datatype** *nat* $=$ *Zero* $({''}0{''})$ | *Suc nat* is the recursively defined datatype of natural numbers, and **datatype** *'a list* $=$ *Nil* | *Cons 'a ''a list''* is both recursive and polymorphic. In application, each type of variable can be replaced by an arbitrary existing datatype.

The structure of a datatype can be represented by an AST in Fig. 3. The AST includes all semantic information of the datatype: the type name, optional type variables and a number of construction rules. Each construction rule contains one constructor and several argument types. These rules specify how a value of the datatype is built from these constructors: a value can be constructed by one constructor with its corresponding construction arguments.

In Isabelle2Cpp, C++ classes or class templates are generated to express datatypes in Isabelle/HOL: a datatype without type variables is expressed by a C++ class, while a polymorphic datatype with type variables is expressed by a C++ class template where the template parameters refer to their relevant type variables. The class (class template) name and template parameters can be generated according to its corresponding AST. In the class (class template), a C++ nested class is generated to express each construction rule: the name of the nested class expresses the corresponding constructor and the types of its data members express the argument types. The C++ class template std::variant represents a type-safe union. An instance of std::variant at any given time either holds a value or one of its alternative types. As a

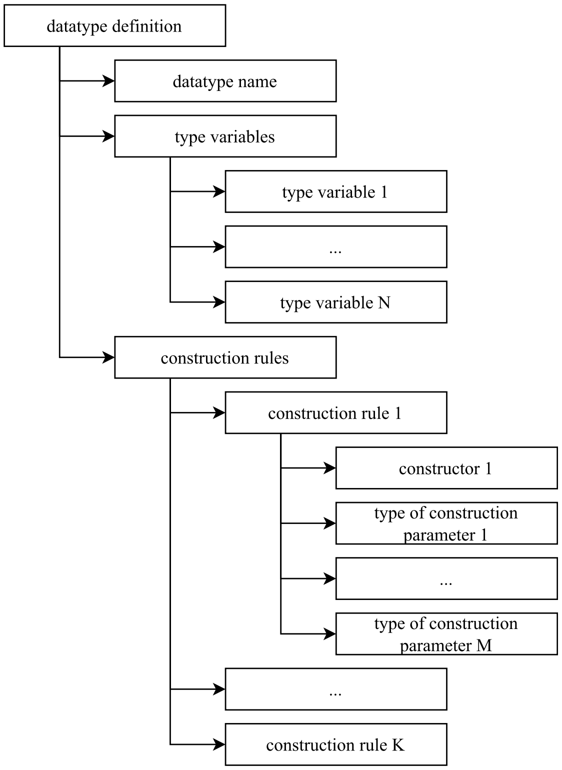

Fig. 3. Structure of a datatype's AST.

value of a datatype is constructed by one datatype construction rule and is represented by its corresponding datatype constructor and relevant construction arguments, a variable of the type std::variant with all nested classes stores the value of this datatype.

The following definition provides a general template C++ class t_c for a non-recursive datatype t.

```
// 'a_1, ..., 'a_n
template<typename T1, ..., typename Tn>
class t {
  // C_1 "τ_{1,1}" ... "τ_{1,n_1}"
  struct _C_1 {
    t_{1,1} p1_;
    ...
    t_{1,n_1} pn_1_;

    const t_{1,1} &p1() const { return p1_; }
    ...
    const t_{1,n_1} &pn_1() const { return pn_1_; }
  };
  ...

  std::variant<_C_1, ..., _C_k> value_;

public:
  t() = default;

  // static functions to construct values of the type
  static t C_1(t_{1,1} p1, ..., t_{1,n_1} pn_1) {
    return t { _C1  p1, ..., pn_1  };
  }
  ...

  // methods to check the value is construted by which constructor
  bool is_C_1() const { return std::holds_alternative<_C_1>(value_); }
  ...

  // methods to read the value as the expected rule
  const _C_1 &as_C_1() const { return std::get<_C_1>(value_); }
  ...
};
```

Here, template parameters T1, ..., Tn are generated to express type variables $'a_1, \ldots, 'a_n$; the nested classes struct $\_C_1$ { $t_{1,1}$ p1_; ...; t_{1,n_1} pn_1_; }; is generated to express the construction rules C_1 $''\tau_{1,1}'' \ldots ''\tau_{1,n_1}''$. In application, the actual value of datatype $('a_1, \ldots, 'a_n)t$ is always stored in the data member value_ of type std:: variant <_C_1, ..., _C_k >. It should be noted that the template header won't be included if the definition of *t* has no type variables. The remaining C++ methods are provided for convenience when the type is used in functions.

For a recursive datatype *t* in Isabelle/HOL, at least one of its construction rules contains *t* itself. An indirect scheme is introduced here to express the recursive datatype due to the limitation of incomplete types in C++. If an argument type in one construction rule is the same with the recursive *t* itself, a smart pointer std:: shared_ptr [17] which points to the type t_c is the generated C++ type of the corresponding data member, which is used to implement the recursion in the construction rule; if an argument type does not contain *t* itself, the generation rule is the same with the rule for non-recursive datatypes. Isabelle2Cpp uses a smart pointer which can automatically allocate and deallocate the memory. In the construction function, std:: make_shared is used to allocate the memory to the pointer. The member is dereferenced when accessing its value. Take **datatype** *snat* = *sZero* | *sSucc snat* as an example, the generated nested struct _sSucc for the construction rule *sSucc snat* and the static construction function snat::sSucc are given as follows:

```
struct snat {
  ...
  struct _sSucc {
    std::shared_ptr<snat> p1_;

    snat p1() const { return *p1_; }
  };
  ...
  static snat sSucc(snat p1) {
    return snat { _sSucc { std::make_shared<snat>(p1) } };
  }
  ...
};
```

The above framework is a strict datatype conversion approach based on datatype definitions. All symbols in a datatype definition are converted into the data members of the relevant C++ class or nested classes. The value of a datatype is converted into an instance of the C++ class. Thus, the structures of the types and their relevant values remain the same.

After the type conversion, the generated C++ class is still defined by the original symbols. In order to use the values of generated C++ class, one can directly access its fields by calling corresponding methods, and may need to define separate C++ functions for interpreting the symbolic values according to his specific purpose.

4.2. *Function conversion*

A function in Isabelle/HOL describes the relationship between its inputs and output. Functions on finite inputs can be defined by enumerating the relationship between its input and output, while functions on infinite recursive datatypes need to be defined by using construction rules. Applying a function to arguments is called evaluation, which substitutes symbols based on the construction rules defined by the function. In Isabelle/HOL, the definition of a function f satisfies the following syntax [18]:

$$\begin{aligned}
&\textbf{fun}\ f\ ::\ {''}t_1 \Rightarrow \ldots \Rightarrow t_{\mathrm{N}} \Rightarrow t_{\mathrm{R}}'' \ \textbf{where}\\
&\quad {''}\mathit{pattern}_1 = \mathit{expression}_1''\\
&\quad \vdots\\
&\quad {''}\mathit{pattern}_{\mathrm{n}} = \mathit{expression}_{\mathrm{n}}''
\end{aligned}$$

Here, f is the name of the function, $t_1, \ldots, t_{\mathrm{N}}$ are the types of the parameters, t_{R} is the type of return value and ${''}\mathit{pattern}_1 = \mathit{expression}_1'', \ldots, {''}\mathit{pattern}_{\mathrm{n}} = \mathit{expression}_{\mathrm{n}}''$ are the equations used to define the function. Pattern matching is used in these equations to define the specific operations on different cases of inputs: as a pattern is a combination of one constructor and some sub-patterns, each pattern is corresponding to one or several construction rules of the parameter types; each expression is a combination of basic operations of the relevant types and calls of functions that have already been defined. In the execution of an Isabelle/HOL function, when a parameter matches a certain pattern, the expression in the corresponding equation is evaluated.

Take the function *add* as an example:

$$\begin{aligned}&\textbf{fun}\ add\ ::\ ''nat \Rightarrow nat \Rightarrow nat''\ \textbf{where}\\ &\quad ''add\ 0\ n = n''\\ &\quad ''add\ (Suc\ m)\ n = Suc\,(add\ m\ n)''\end{aligned}$$

The function with the name *add* has two *nat* input parameters and one *nat* output, and it describes the relationship between the inputs and the output by the construction rules $''add\ 0\ n = n''$ and $''add\ (Suc\ m)\ n = Suc\,(add\ m\ n)''$. In the rule $''add\ (Suc\ m)\ n = Suc\ (add\ m\ n)''$, $add\ (Suc\ m)\ n$ is a pattern for inputs, while the expression $Suc\,(add\ m\ n)$ is for the corresponding output.

In the function conversion, an AST that includes the name, type and equations is used as the IR between the Isabelle/HOL specification and the relevant C++ code (see Fig. 4). The corresponding C++ function's name, type and body will be generated from the AST.

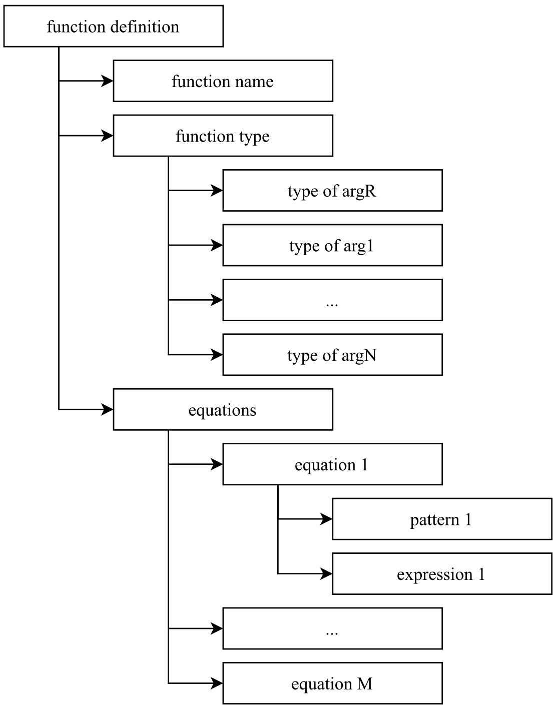

Fig. 4. Structure of a function's AST.

Compared with the functional Isabelle/HOL, C++ programs are mainly written in an imperative approach: An expression in C++ is a sequence of operators and their operands, and the evaluation of an expression leads to a result; a statement is a compositions of control-flow and expressions; a function is an encapsulation and abstraction of a sequence of statements. Therefore, the key of function conversion lies in obtaining statements that describe the computation process from inputs to outputs according to the equations in Isabelle/HOL.

In the function conversion, types $t_1, \ldots, t_{\mathrm{N}}$ of parameters and the type t_{R} of the return value are converted into C++ types according to the type conversion; then the corresponding C++ types are used directly in the definition of the corresponding C++ function. In C++, operations performed on parameters require specific variable names to distinguish different relevant parameters. Therefore, the name arg$i$ is automatically generated to express the ith parameter of the Isabelle/HOL function. Then, the general form of the generated C++ function is as follows:

```
t_Rc f(t_1c arg1, ..., t_Nc argN) {
  ...
}
```

Furthermore, if the type of f in Isabelle/HOL contains type variables, a function template will be used

```
template<typename T1, ..., typename Tk> t_Rc f(t_1c arg1, ..., t_Nc argN) {
  ...
}
```

Here, k is the number of different type variables in the definition of f.

It should be noted that sometimes the parameter of a function may have a function type. Then the function type will be converted into an instance of the class template std::function. For example, a function *app* with a type of ″nat ⇒ (nat ⇒ nat) ⇒ nat″ will generate the following target C++ function:

```
nat app(nat arg1, std::function<nat(nat)> arg2) {
  ...
}
```

4.2.1. *Pattern conversion*

Isabelle/HOL uses pattern matching to distinguish different cases of the function inputs. Different expressions are returned according to the inputs' constructors. In the function conversion, compound if-statements are generated for pattern matching: the pattern of an equation is converted into conditions and optional declarations, and the expression is converted into a final return-statement and some optional auxiliary statements. Thereby, each equation of the Isabelle/HOL function is converted into an if-statement block with optional nested if-statements in C++. An if-statement block corresponding to an equation is called a Conditional Statements Module (CSM) in this paper. When generating the if-statements, parameters are decomposed recursively to generate optional conditions and declarations: a condition is generated when a new pattern appears; a variable declaration will be generated in the innermost if-statement if there is an identifier that is not a constructor in $pattern_i$. After the generation of the conditions and declarations, optional auxiliary statements [statements...] and return-statement are generated to return the expression of the equation in the innermost if-statement. The CSMs generated for each equation $''pattern_i = expression_i''$ have the following form:

```
t_Rc f(...) {
  // CSM1: "pattern_1 = expression_1"
  if (condtion_1) {
    ...
    if (condition_m) {
      [declarations...]

      [statements...]
      return expression_1c;
    }
  }
  ...
  // CSMn: "pattern_n = expression_n"
  if (...) {
    ...
  } else {
    abort();
  }
}
```

Here condition$_1$, ..., condition$_m$ are the conditions to check whether the construction of parameters satisfies $pattern_1$. If these conditions are satisfied, the corresponding statements will be executed and the expression$_{1c}$ will be returned; otherwise, the next pattern will be checked. It should be noted that the implementation of pattern checking is part of the generated C++ type class, where the method "is_C$_i$" is generated to check whether the corresponding pattern is satisfied.

If "C$_i$ x$_1$ ... x$_N$" is one construction rule of type t, then the following conversion procedure is applied in the function generation:

```
$curr.is_C_i()
%match x_1 $curr.as_C_i().p1()
...
%match x_N $curr.as_C_i().pN()
```

Here "\$curr" refers to the generated C++ expression corresponding to the current pattern. The expression "\$curr.is_C_i()" is the first generated condition for the pattern "C_i x_1 ... x_N" and the followed statements "%match x_i e_i" represents the recursive generation of a sub-condition, which determines whether a C++ expression e_i matches a sub-pattern x_i. In the recursive generation of sub-patterns, "\$curr. as_C_i.pj" represents the corresponding C++ expression e_i and is used to visit the jth argument of the ith construction rule.

4.2.2. *Expression conversion*

After the generation of declarations and conditions for pattern matching, the C++ statements corresponding to the Isabelle/HOL expression of equations are generated in the innermost if-statement. When constructing the expression of an return-statement in C++, the sub-expressions are generated recursively. This process may introduce some other statements besides the return-statement to assist the expression construction.

In the conversion, as some expressions in Isabelle/HOL depend on the definitions of types, such as "*Cons* x_1 x_2" of type *'a list*, the conversion for these expressions depends on the definitions of their relevant types. For expressions that do not depend on types, they can be converted directly. Therefore, in this paper, expressions in Isabelle/HOL are classified as: value constructions, variables, function calls and control-flow expressions such as If-expression, Case-expression, Let-expression and Lambda-expression.

A value construction in Isabelle/HOL includes one constructor and some construction arguments of the argument types in the corresponding construction rule. If a datatype t has a construction rule "C_k τ_1 ... τ_N", then "C_k x_1 ... x_N" is the value corresponding to this construction rule in Isabelle/HOL, where x_i is the ith argument of the type τ_i. In the conversion, "C_k x_1 ... x_N" is converted into the C++ expression "t::$\mathrm{C}_\mathrm{k}(\mathrm{x}_{1\mathrm{c}}, \ldots, \mathrm{x}_{\mathrm{Nc}})$" accordingly, where $\mathrm{x}_{i\mathrm{c}}$ is the generated C++ expression of x_i.

A variable in Isabelle/HOL introduced in a pattern is more an identifier than a constructor. The corresponding C++ variable declaration statement is generated when processing the pattern. Therefore, the variable name is directly used as the corresponding variable in the corresponding generated C++ expression.

A function call "f $\mathrm{x}_1 \ldots$ x_N" in Isabelle/HOL is converted into the C++ function call expression "$\mathrm{f}(\mathrm{x}_{1\mathrm{c}}, \ldots, \mathrm{x}_{\mathrm{Nc}})$", where $\mathrm{x}_{i\mathrm{c}}$ is the generated C++ expression corresponding to the ith argument x_i of the function call in Isabelle/HOL.

An If-expression "*If cond true_expr false_expr*" in Isabelle/HOL cannot be converted into the ternary operation <> ? <>:<> in C++ directly since that assistant statements may be required in the generation of expressions "*cond*", "*true_expr*" and "*false_expr*". Therefore, for an If-expression "*If* x_1 x_2 x_3" of type t, the following C++ expression is generated:

```
t_c $temp;
if (x_1c) {
  $temp = x_2c;
} else {
  $temp = x_3c;
};
temp
```

A Case-expression in Isabelle/HOL is used to analyze elements of a datatype, which has the following form:

$$\texttt{case}\ e\ \texttt{of}\ \mathit{pattern}_1 \Rightarrow e_1 \mid \ldots \mid \mathit{pattern}_\mathrm{m} \Rightarrow e_\mathrm{m}$$

This expression is evaluated as e_i if e is of the form $\mathit{pattern}_i$. For a Case-expression, sub-CSMs are generated in the body of a C++ lambda expression like a function. The generated C++ expression for the Case-expression is a call of the generated C++ lambda expression. The generated C++ code has the following form:

```
auto $temp0 = ([&] {
  auto $temp1 = e_c;

  // pattern_1 ⇒ e_1
  if (condtion_1) {
    [declarations...]
    ...
    if (condition_m) {
      [declarations...]

      [statements...]
      return e_1c;
    }
  }
  ...
})();
$temp0
```

A Let-expression in Isabelle/HOL is used to support local definitions. For example, the value of an expression "*let* $x = t$ *in* u" is equivalent to the value of the expression u after replacing all free occurrences of x with t. In general, a Let-expression has the following form:

$$\texttt{let}\ x_1 = t_1\,\texttt{;}\ \ldots\ \texttt{;}\ x_\mathrm{n} = t_\mathrm{n}\ \texttt{in}\ u$$

In the conversion, each pair of x_i and t_i is converted into a local variable declaration. Then the whole Let-expression is converted into the C++ expression u_c, which is the

C++ expression corresponding to the Isabelle/HOL expression u. The C++ expression for a Let-expression is as follows:

```
auto x1 = t1c;
...
auto xn = tnc;
uc
```

A Lambda-expression in Isabelle/HOL is performed as a function with parameters and an expression as the return value, which has the following form:

$$\lambda x_1 \ \ldots \ x_{\mathrm{n}}.\, t$$

where $x_1, \ldots, x_{\mathrm{n}}$ are parameters and t is the return value. A Lambda-expression is converted into a C++ lambda expression of the following form:

```
[&] (T1 x1, ..., Tn xn) { return tc; }
```

where T1, ..., Tn are types of $x_1, \ldots, x_{\mathrm{n}}$. If a Lambda-expression does not have type information, T1, ..., Tn are generated as the type placeholders and require users to input corresponding types.

The definition-based conversion uses isomorphic structures and semantically equivalent operations to generate C++ types and functions from Isabelle/HOL datatypes and functions. In other words, the structures of types and functions in Isabelle/HOL and C++ are the same which guarantees the construction rules in Isabelle/HOL datatypes are well reflected in the generated C++ classes (class templates); furthermore, the relationships between inputs and outputs of Isabelle/HOL functions are converted into sequences of semantically equivalent statements in the generated C++ functions (function templates). The carefully designed isomorphic structures and semantically equivalent operations lay the foundations for the correctness of the definition-based conversion.

5. Rule-Based Conversion

In Isabelle/HOL and C++, there are some existing types that express the same concepts but have different implementations. In Isabelle/HOL, commonly used algebraic datatypes, such as *bool* of Booleans, *nat* of natural numbers, *int* of integers, and the polymorphic datatype *'a list* of linked lists are defined by enumeration or recursion [19]. In C++, arithmetic types, such as the type $\mathrm{bool_c}$ of Booleans, the type $\mathrm{int_c}$ of integers and the type unsigned $\mathrm{int_c}$ of natural numbers, as well as class templates std::list of linked lists are represented by fixed-width binary sequences.

For these datatypes, applying the above definition-based conversion method may arise low efficiency of the generated C++ codes, because the generated C++ codes depend on the specific constructor of each value. In order to improve the execution efficiency of the generated C++ codes and make full use of the existing types, this paper further proposes a rule-based conversion for existing types that include Isabelle/HOL number types and other commonly used datatypes. This conversion is performed according to concrete conversion rules under certain conditions, which are provided in a semantically equivalent correspondence table of types and operations. The process of the rule-based conversion is shown in Fig. 5.

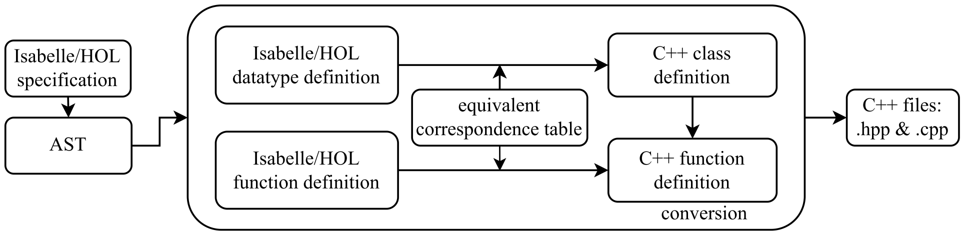
Isabelle/HOL specification
AST
Isabelle/HOL datatype definition
Isabelle/HOL function definition
equivalent correspondence table
C++ class definition
C++ function definition
conversion
C++ files: .hpp & .cpp

Fig. 5. The rule-based conversion process.

5.1. *Type conversion*

Isabelle/HOL number types always refer to the datatypes defined based on mathematical definitions in Isabelle/HOL whose ranges are unlimited such as the type *nat* of natural numbers, the type *int* of integers, etc. On the one hand, these equivalent recursive definitions can be achieved by the definition-based conversion, but it will cause to much performance loss during calculations, much slower than calculation with fixed-width types, such as unsigned int_c, int_c in C++. On the other hand, if the calculation does not overflow or the calculation is only about comparison, fixed-width types are enough to guarantee the correctness.

Therefore, this paper converts these Isabelle/HOL number types (i.e. *nat* and *int*) into std::uint64_t and std::int64_t to improve the execution efficiency of the generated C++ codes. The basic operations on these datatypes, such as addition and multiplication, can be directly converted too. Table 1 lists the conversion for *nat*, *int* and corresponding operators.

Besides, other commonly used types such as *bool*, *list*, *set* and *option* pre-defined in Isabelle/HOL and in C++ types such as bool_c, std::list, std::set and std::optional are used to describe the same concepts, which do not involve the underlying representation. Thus, the targets of these Isabelle/HOL types are set as the corresponding

Table 1. Conversion for *nat*, *int* and corresponding operators.

Isabelle/HOL type	C++ type	Supported Isabelle/HOL Operators
nat	std::uint64_t	$+, -, *, /, div, mod, \hat{}, =, \neq, \leq, <, \geq, >$
int	std::int64_t	$+, -, *, /, div, mod, \hat{}, =, \neq, \leq, <, \geq, >$

Table 2. Conversion for commonly used Isabelle/HOL types and corresponding expressions.

Isabelle/HOL type	C++ type	Supported Isabelle/HOL Expressions
bool	bool_c	$\mathit{True}, \mathit{False}, x_1 \wedge x_2, x_1 \vee x_2, x_1 = x_2, x_1 \neq x_2$
'a list	std::list<T>	$\mathit{Nil}, \mathit{Cons}\ x_1\ x_2, [x_1, \ldots, x_N], x_1 @ x_2, x_1 \# x_2, \ldots$
'a set	std::set<T>	$\{x\dot{1}, \ldots, x_N\}, x_1 \cap x_2, x_1 \cup x_2, x_1 \in x_2, \ldots$
'a option	std::optional<T>	$\mathit{Some}\ x_1, \mathit{None}$
*'a * 'b*	std::pair<T1, T2>	$\mathit{Pair}\ x_1\ x_2, (x_1, x_2), \mathit{fst}\ x_1, \mathit{snd}\ x_1$

C++ types directly. Table 2 lists the conversion for these types and corresponding expressions, where x_i represents the ith sub-expression from left to right, and the specific conversion of expressions is provided in Sec. 5.2.

It should be noted that the conversion from the datatypes *nat* and *int* in Isabelle/HOL to the C++ types std::uint64_t and std::int64_t may cause overflow risks in the generated C++ codes. The value ranges of these types in the two languages are different. A value of datatype *nat* or *int* can represent a number with a countable infinite range due to their recursive definitions in Isabelle/HOL, while a value of type std::uint64_t or std::int64_t in C++ represented by 64 bits that can only represent 2^{64} distinct values. An overflow will occur when there is an operation producing a value out of the range. This will lead to inconsistent behaviors between the generated C++ functions and the original Isabelle/HOL specifications. To guarantee the correctness, the definition-based conversion should be used if there are overflow risks.

Isabelle/HOL also provides theories for machine words which include the type *'a word* that can be used to formalize the fixed-width types in C++ [20]. The operational semantics of *'a word* in Isabelle/HOL and fixed-width types in C++ are the same. Therefore, if strict boundary attributes are required, the conversion from *'a word* to the corresponding C++ fixed-width types can be applied. For these cases, it only needs to specify the width as required when type *'a word* such as 8 *word*, 32 *word*, is used in the relevant specifications of Isabelle/HOL, then the generation of C++ fixed-width types from Isabelle/HOL word is also realized by the correspondence table. Table 3 lists the conversion for *'a word* and corresponding operators.

Table 3. Conversion for *'a word* and corresponding operators.

Isabelle/HOL type	C++ type	Supported Isabelle/HOL Operators
N *word* $(\mathrm{N} = 8, 16, 32, 64)$	std::uintN_t	$+, -, *, /, \mathit{div}, \mathit{mod}, \hat{}, =, \neq, \leq, <, \geq, >$

For convenience, in what follows, the pre-defined types are used to stand for the datatypes that need to be converted based on rules. Similarly, the user-defined types are used to represent the datatypes that need to be converted based on their definitions.

5.2. *Operation conversion*

For the patterns which are relevant to the types converted based on rules, the conversion rules from patterns to conditions are shown in Table 4, where the meanings of the symbols in the condition conversion procedure are the same as above in Sec. 4.2.2. Furthermore, in "%match x_i e_i", e_i may have some other statements in the beginning to help generate the corresponding C++ expression. For example, if e_i is "auto $temp0 = curr.front(); $temp0", where "$temp0" represents a generated non-repeated temporary variable, then the statement "auto $temp0 = curr.front();" will be generated before the if-statement for pattern x_i, and "$temp0" will be the expression to match x_i.

Table 4. Conversion rules from Isabelle/HOL patterns to C++ conditions.

Isabelle/HOL type	Pattern	C++ Condition conversion procedure
nat	$n(n = 0, 1, \ldots)$	$curr == n
∼	*Zero*	$curr == 0
∼	*Suc* x_1	$curr! = 0
	$\hookrightarrow$	%match x_1 $curr - 1
int	$n(n = \ldots, -1, 0, 1, \ldots)$	$curr == n
bool	*True*	$curr == true
∼	*False*	$curr == false
'a'list	*Nil*	$curr.empty ()
∼	*Conx* $\dot{x1}$ $\dot{x2}$	!$curr.empty ()
	$\hookrightarrow$	%match x_1 $curr.front ()
	$\hookrightarrow$	%match x_2 auto $temp0 = decltype ($curr) {std::next($curr.begin(), $curr.end())}; $temp0
∼	[]	// *same as "Nil"*
∼	x_1 # x_2	// *same as "Conx* x_1 x_2"
'a set	{}	$curr.empty ()
'a option	*None*	!$curr.has_value ()
∼	*Some* x_1	$curr.has_value ()
	$\hookrightarrow$	%match x_1 $curr.value ()
$'a * 'b$	*Pair* x_1 x_2	%match x_1 $curr.first
	$\hookrightarrow$	%match x_2 $curr.second
∼	(x_1, x_2)	// *same as "Pair* x_1 x_2"
N *word* $(N = 8, 16, 32, 64)$	$n(n = 0, \ldots, 2^N - 1)$	$curr == n

The expressions, which are relevant to the types converted based on rules, include related value constructions, function calls and use of specific operators, e.g. *Zero* and + for *nat*, *Nil* and @ for *list*. In order to avoid inefficiency caused by recursion, recursive operations of these types in Isabelle/HOL are converted directly into C++ expressions with assistant statements if they have equivalent operational semantics.

For implementation, they are converted into C++ expressions, which generally have a form of the combinations of generated C++ sub-expressions with assistant statements. A generated C++ expression with assistant statements is called an expression-sequence in this paper, which has the form "stmt_0; …; stmt_N; expr". Every expression in Isabelle/HOL is a new entity, while entities (e.g. a std::list) in C++ rely on variables. Therefore, the construction of new entities in C++ consists of variable declarations and relevant operations on the variables, where assistant statements are needed.

For example, Isabelle/HOL value construction "*Cons* x_1 x_2" is converted into a C++ expression-sequence "std::list<T> \$temp = x_{2c}; \$temp.push_front(x_{1c}); \$temp", where "std::list<T>\$temp = x_{2c};" and "\$temp.push_front(x_{1c});" are assistant statements and "\$temp" is the generated C++ expression corresponding to the Isabelle/HOL expression "*Cons* x_1 x_2". Table 5 provides some conversion rules from

Table 5. Conversion rules from Isabelle/HOL expressions to C++ expression-sequences.

Isabelle/HOL type	C++ type	Isabelle/HOL Expression	C++ Expression-sequence
nat	std::uint64_t	$n (n = 0, 1, \ldots)$	n
~	~	*Zero*	0
~	~	*Suc* x_1	$x_{1c} + 1$
~	~	$x_1 + x_2$	$(x_{1c}) + (x_{2c})$
…	…	…	…
int	std::int64_t	$n (n = \ldots, -1, 0, 1, \ldots)$	n
~	~	$x_1 + x_2$	$(x_{1c}) + (x_{2c})$
…	…	…	…
bool	bool_c	*True*	true
~	~	*False*	false
~	~	$x_1 \wedge x_2$	$(x_{1c})\ \ (x_{2c})$
…	…	…	…
'a list	std::list<T>	*Nil*	std::list<T> \$temp ()
~	~	*Cons* x_1 x_2	std::list<T>\$temp = x_{2c};
		$\hookrightarrow$	\$temp.push_front(x_{1c});
		$\hookrightarrow$	\$temp
~	~	[]	// *same as "Nil"*
~	~	x_1 # x_2	// *same as "Conx* x_1 x_2*"*
~	~	$[x_1, x_2, \ldots, x_N]$	std::list<T>$\{x_{1c}, x_{2c}, \ldots, x_{Nc}\}$
~	~	x_1 @ x_2	std::list<T>\$temp0 = x_{1c};
		$\hookrightarrow$	std::list<T>\$temp1 = x_{2c};
		$\hookrightarrow$	\$temp0.splice(\$temp0.end(), \$temp1);
		$\hookrightarrow$	\$temp0
~	~	*length* x_1	x_{1c}.size()
…	…	…	…
'a set	std::set<T>	{}	std::set<T> ()
~	~	$\{x_1, x_2, \ldots, x_N\}$	std::set<T>$\{x_{1c}, x_{2c}, \ldots, x_{Nc}\}$
~	~	$x_1 \cap x_2$	auto \$temp0 = x_{1c};
		$\hookrightarrow$	auto \$temp1 = x_{2c};
		$\hookrightarrow$	\$temp0.merge(\$temp1);
		$\hookrightarrow$	\$temp1
…	…	…	…
'a option	std::optional<T>	*Some* x_1	std::make_optional<T> (x_{1c})
~	~	*None*	std::optional<T> ()
*'a * 'b*	std::pair<T1, T2>	*Pair* x_1 x_2	std::make_pair<T1, T2> (x_{1c}, x_{2c})

Table 5. (*Continued*)

Isabelle/HOL type	C++ type	Isabelle/HOL Expression	C++ Expression-sequence
∼	∼	*fst* x_1	x_{1c}.first
∼	∼	*snd* x_1	x_{1c}.second
∼	∼	(x_1, x_2)	// *same as* "*Pair* x_1 x_2"
N *word* $(N = 8, 16, 32, 64)$	std::uintN_t	$n (n = 0, \ldots, 2^N - 1)$	n
∼	∼	$x_1 + x_2$	$(x_{1c}) + (x_{2c})$
...	...	...	...

expressions, which are relevant to types converted based on rules, in Isabelle/HOL to C++ expression-sequences.

Compared with the previous definition-based conversion method, this rule-based method improves the execution efficiency but may reduce the correctness of the generated C++ codes. For a more detailed comparison, the definition-based conversion performs in a way similar to that of the code generator of Isabelle/HOL. The definition-based conversion parses the specifications into ASTs, which are then converted into semantically equivalent C++ classes (class templates) and functions (function templates) by using the same symbols; while the code generator of Isabelle/HOL transforms the equations in the specification into an equivalent intermediate language program, and then serializes the intermediate program to the final implementation of the target languages [21]. In the rule-based conversion, efficiency is considered and conversion is performed based on the common concepts where concrete conversion rules are manually provided. This approach is similar to that of data refinement in Isabelle/HOL. The rule-based conversion is more flexible and usually makes the generated C++ codes more efficient.

6. Case Studies

Based on Isabelle2Cpp, this paper implements automatic C++ code generation of Isabelle/HOL, which includes some commonly used types (such as *bool*, *nat*, *int*, *pair*, *option*, *list*, *set*), user-defined types and function specifications. In this section, two examples are used to illustrate why Isabelle2Cpp is feasible: the first is the conversion of two merge sort algorithm specifications in Isabelle/HOL; the second is the conversion of the Isabelle/HOL theory List.thy which includes 78 definitions.

6.1. *Merge sort algorithm*

In order to illustrate the feasibility of the definition-based conversion and the rule-based conversion, two specifications of the merge sort algorithm in Isabelle/HOL are converted into C++ code: one is written with pre-defined Isabelle/HOL type *nat list* and the other is written with user-defined datatypes.

6.1.1. *Specification with predefined types*

In Isabelle/HOL, the specification of merge sort algorithm for *nat list* is defined as follows, whose correctness is proved in [22]:

```
fun merge :: "nat list \<Rightarrow> nat list \<Rightarrow> nat list"
↪  where
  "merge xs [] = xs" |
  "merge [] ys = ys" |
  "merge (x # xs) (y # ys) = If (x \<le> y) (x # (merge xs (y # ys))) (y #
  ↪  (merge (x # xs) ys))"

fun merge_sort :: "nat list \<Rightarrow> nat list" where
  "merge_sort [] = []" |
  "merge_sort [x] = [x]" |
  "merge_sort xs = merge (merge_sort (take ((length xs) div 2) xs))
  ↪  (merge_sort (drop ((length xs) div 2) xs))"
```

The generated C++ code is as follows. In this rule-based conversion, *nat list* is converted into std::list<std::uint64_t>, and each equation of the function specifications *merge* and *merge_sort* corresponds to a CSM. For example, the first equation *"merge xs* []*"* of the function *merge* is converted into the CSM on lines 2 to 3. The functions (i.e. *take*, *length* and *drop*) and operators (i.e. [], , ≤, etc.) have semantic equivalent operations in C++. Therefore, they can be converted directly according to corresponding conversion rules, which have been added to the correspondence table in Table 5. It can be seen that the procedure of the generated C++ function is consistent with the Isabelle/HOL function specification, which guarantees the correctness of the C++ code.

```
1  std::list<std::uint64_t> merge(const std::list<std::uint64_t> &arg1, const
   ↪  std::list<std::uint64_t> &arg2) {
2    // merge xs [] = xs
3    if (arg2.empty()) { return arg1; }
4
5    // merge [] ys = ys
6    if (arg1.empty()) { return arg2; }
7
8    // merge (x # xs) (y # ys) = If (x \<le> y) (x # (merge xs (y # ys))) (y
     ↪  # (merge (x # xs) ys))
```

```
  auto x = arg1.front();
  auto xs = decltype(arg1){std::next(arg1.begin()), arg1.end()};
  auto y = arg2.front();
  auto ys = decltype(arg2){std::next(arg2.begin()), arg2.end()};
  std::list<std::uint64_t> temp0;
  if (x <= y) {
    auto temp1 = ys;
    temp1.push_front(y);
    auto temp2 = merge(xs, temp1);
    temp2.push_front(x);
    temp0 = temp2;
  } else {
    auto temp3 = xs;
    temp3.push_front(x);
    auto temp4 = merge(temp3, ys);
    temp4.push_front(y);
    temp0 = temp4;
  }
  return temp0;
}

std::list<std::uint64_t> merge_sort(const std::list<std::uint64_t> &arg1)
↪  {
  // merge_sort [] = []
  if (arg1.empty()) { return std::list<std::uint64_t>(); }

  // merge_sort [x] = [x]
  if (arg1.size() == 1) {
    auto x = *std::next(arg1.begin(), 0);
    return std::list<std::uint64_t>{x};
  }

  // merge_sort xs = merge (merge_sort (take ((length xs) div 2) xs))
  ↪  (merge_sort (drop ((length xs) div 2) xs))
  return merge(merge_sort(decltype(arg1){ arg1.begin(),
  ↪  std::next(arg1.begin(), arg1.size() / 2) }),
  ↪  merge_sort(decltype(arg1){ std::next(arg1.begin(), arg1.size() / 2),
  ↪  arg1.end() }));
}
```

6.1.2. *Specification with user-defined datatypes and functions*

In order to show the feasibility of the definition-based conversion, user-defined linked list datatype *'a slist* is provided. The form of this definition is similar to *'a slist*, but it uses different symbols. The relevant functions *slength*, *stake* and *sdrop* based on the type *'a slist* are defined, respectively, and similar to the definitions of functions *length*, *take* and *drop*. The final specifications of *smerge* and *smerge_sort* are defined based on *'a slist* and the relevant functions.

The definitions of *'a slist*, *smerge* and *smergesort* are as follows:

```
datatype 'a slist = sNil | sCons 'a "'a slist"

fun smerge :: "nat slist \<Rightarrow> nat slist \<Rightarrow> nat slist"
↪  where
  "smerge xs sNil = xs" |
  "smerge sNil ys = ys" |
  "smerge (sCons x xs) (sCons y ys) = If (x \<le> y) (sCons x (smerge xs
  ↪  (sCons y ys))) (sCons y (smerge (sCons x xs) ys))"

fun smerge_sort :: "nat slist \<Rightarrow> nat slist" where
  "smerge_sort sNil = sNil" |
  "smerge_sort (sCons x sNil) = sCons x sNil" |
  "smerge_sort xs = smerge (smerge_sort (stake ((slength xs) div 2) xs))
  ↪  (smerge_sort (sdrop ((slength xs) div 2) xs))"
```

With the help of Isabelle2Cpp, the generated C++ class template slist and the definitions of the functions slength, stake, sdrop, smerge and smerge_sort can be obtained, respectively.

The definition of C++ class template slist is as follows:

```
template<typename T1>
class slist {
  struct _sNil {};
  struct _sCons {
    T1 p1_;
    std::shared_ptr<slist<T1>> p2_;

    const T1 &p1() const { return p1_; }
    slist<T1> p2() const { return *p2_; }
  };

  std::variant<_sNil, _sCons> value_;
  slist(const std::variant<_sNil, _sCons> &value) : value_(value) {}

public:
  slist() = default;

  static slist<T1> sNil() { return slist<T1> { _sNil {} }; }
  static slist<T1> sCons(T1 p1, slist<T1> p2) { return slist<T1> { _sCons
    ↪  { p1, std::make_shared<slist<T1>>(p2) } }; }

  bool is_sNil() const { return std::holds_alternative<_sNil>(value_); }
  bool is_sCons() const { return std::holds_alternative<_sCons>(value_); }

  const _sCons &as_sCons() const { return std::get<_sCons>(value_); }
};
```

In this C++ code, the template parameter T1 is used to describe the type variable *'a* of the datatype *'a slist*; the nested classes struct _sNil {} and struct _sCons { ... } are used to describe the construction rules *sNil* and *sCons 'a "'a slist"*, respectively.

The final C++ functions smerge and smerge_sort are generated as follows:

```
slist<std::uint64_t> smerge(const slist<std::uint64_t> &arg1, const
↪  slist<std::uint64_t> &arg2) {
  // smerge xs sNil = xs
  if (arg2.is_sNil()) { return arg1; }

  // smerge sNil ys = ys
  if (arg1.is_sNil()) { return arg2; }

  // smerge (sCons x xs) (sCons y ys) = If (x \<le> y) (sCons x (smerge xs
  ↪  (sCons y ys))) (sCons y (smerge (sCons x xs) ys))
  auto x = arg1.as_sCons().p1();
  auto xs = arg1.as_sCons().p2();
  auto y = arg2.as_sCons().p1();
  auto ys = arg2.as_sCons().p2();
  slist<std::uint64_t> temp0;
  if (x <= y) {
    auto temp1 = slist<std::uint64_t>::sCons(y, ys);
    auto temp2 = slist<std::uint64_t>::sCons(x, smerge(xs, temp1));
    temp0 = temp2;
  } else {
    auto temp3 = slist<std::uint64_t>::sCons(x, xs);
    auto temp4 = slist<std::uint64_t>::sCons(y, smerge(temp3, ys));
    temp0 = temp4;
  }
  return temp0;
}

slist<std::uint64_t> smerge_sort(const slist<std::uint64_t> &arg1) {
  // smerge_sort sNil = sNil
  if (arg1.is_sNil()) { return slist<std::uint64_t>::sNil(); }

  // smerge_sort (sCons x sNil) = sCons x sNil
  if (arg1.is_sCons()) {
    auto x = arg1.as_sCons().p1();
    if (arg1.as_sCons().p2().is_sNil()) {
      auto temp0 = slist<std::uint64_t>::sCons(x,
      ↪  slist<std::uint64_t>::sNil());
      return temp0;
    }
  }

  // smerge_sort xs = smerge (smerge_sort (stake ((slength xs) div 2) xs))
  ↪  (smerge_sort (sdrop ((slength xs) div 2) xs))
  return smerge(smerge_sort(stake(slength(arg1) / 2, arg1)),
  ↪  smerge_sort(sdrop(slength(arg1) / 2, arg1)));
}
```

The examples of *merge_sort* and *smerge_sort* and corresponding generated C++ code are used to illustrate the differences between the definition-based conversion and the rule-based conversion: first, the definition-based conversion requires the complete definitions of datatypes and functions, while the rule-based conversion requires the mapping rules of types and operations; second, the correctness of the definition-based conversion, which strictly adheres to the formal definition of datatypes and functions, can be fully guaranteed, while the correctness of the rule-based conversion depends on the assumption that the properties of the program before and after conversion are consistent; third, the definition-based conversion will include all recursive operations of the specification in Isabelle/HOL, while the rule-based conversion may introduce more assignments for efficiency. For sorting, no overflow is introduced since it only involves comparison instead of calculation, thus both results of the conversions are correct. To show the correctness of the generated codes, test cases are randomly generated and run in Isabelle/HOL and C++ respectively. The results of all the tests in the two languages are both correct and consistent. Besides, the correctness of the generated C++ codes has been manually verified; the consistency of the Isabelle/HOL specifications and the generated C++ codes has been checked.

6.2. *HOL/List.thy*

In order to illustrate the applicability of Isabelle2Cpp, the raw file HOL/List.thy in Isabelle2020 is processed by Isabelle2Cpp. The framework outputs C++ files List.hpp and List.cpp without manual work. Class definitions and function declarations are included in List.hpp and function definitions are in List.cpp.

There are 78 Isabelle/HOL definitions, one datatype and 77 functions, in the List.thy. 18 of the functions that include set operations are ignored because the set in

Table 6. Results and failure reasons of converting List.thy.

Definitions in List.thy	Converted	Note
(44) *'a list, last, butlast, append, rev, filter, foldl, concat, drop, take, nth, list_update, takeWhile, dropWhile, zip, upt, insert, find, count_list, remove1, removeAll, distinct,remdups, remdups_adj, replicate, length,enumerate, rotate1, splice, min_list, arg_min_list, insort_key, partition, upto, upto_aux, listset, bind, map_tailrec_rev, map_tailrec, member, list_ex, null, maps, gen_length*	yes	
(18) *coset, sorted_wrt, sorted, strict_sorted, lex, lenlex, lexord, listrell, list_ex1, can_select, listrellp, lexordp, set_Cons, lexn, all_interval_nat, all_interval_int, map_project*	no	usage of set
(16) *rotate, fold, foldr, sort_key, subseqs, those, product, union, n_lists, nths, insort_insert_key, map_filter, map2, product_lists, extract, sorted_list_of_set*	no	usage of advanced functional features and lack of type information

Isabelle/HOL involves the axiom of choice and may be uncountable, and non-countable sets cannot be handled in C++ directly.

After the conversion, 44 of the rest 60 definitions in the List.thy are successfully converted into C++ codes (611 lines) automatically. The generated C++ codes are manually verified and all codes pass the designed tests. The results and failure reasons of converting List.thy are shown in Table 6.

The reasons that 16 definitions cannot be converted lie in the fact that: These function definitions use functional features such as currying of functions and closures that lack type information, and need type inference. These definitions can be converted by this framework if the currying and uncurrying of functions and the type inference are implemented.

7. Conclusion

This paper provides a conversion framework Isabelle2Cpp to convert Isabelle/HOL specifications into C++ codes. Written in C++, Isabelle2Cpp includes over 3,100 lines of codes. With this framework, existing Isabelle/HOL specifications can be used for reliable generation of executable C++ programs. The correctness of the generated C++ codes is guaranteed by three aspects: (i) the correctness of Isabelle/HOL specification, which can be formally verified in Isabelle/HOL; (ii) the correct implementation of the framework, whose structure is clear and the implementation can be easily checked and tested; (iii) the equivalence of operations in conversion rules: for the definition-based conversion, it is accomplished according to the strict mapping rules of symbols and structures, which guarantees the semantic equivalence; for the rule-based conversion, the semantic equivalence is guaranteed by certain conditions, where users should take notice. Thus, once a user chooses the appropriate conversion way, he/she can obtain the reliable C++ codes accordingly.

Acknowledgments

This work was supported by Fundamental Research Funds for Central Universities under Grant No. BLX2015-17.

References

1. T. Nipkow, L. C. Paulson and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Vol. 2283 (Springer Science & Business Media, 2002).
2. M. Wenzel, L. C. Paulson and T. Nipkow, The Isabelle framework, in *Int. Conf. Theorem Proving in Higher Order Logics*, 2008, pp. 33–38.
3. F. Haftmann and T. Nipkow, A code generator framework for Isabelle/HOL, in *Theorem Proving in Higher Order Logics: Emerging Trends Proc.*, 2007, p. 121.
4. F. Haftmann, A. Krauss, O. Kunčar and T. Nipkow, Data refinement in Isabelle/HOL, in *Int. Conf. Interactive Theorem Proving*, 2013, pp. 100–115.

5. L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök and J. Matthews, Imperative functional programming with Isabelle/HOL, in *Int. Conf. Theorem Proving in Higher Order Logics*, 2008, pp. 134–149.
6. S. L. P. Jones and P. Wadler, Imperative functional programming, in *Proc. 20th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, 1993, pp. 71–84.
7. P. Lammich, Refinement to imperative HOL, *J. Autom. Reason.* **62**(4) (2019) 481–503.
8. P. Lammich, Generating verified LLVM from Isabelle/HOL, in *10th Int. Conf. Interactive Theorem Proving*, 2019, pp. 1–19.
9. F. Mehta and T. Nipkow, Proving pointer programs in higher-order logic, in *Int. Conf. Automated Deduction*, 2003, pp. 121–135.
10. D. Greenaway, J. Andronick and G. Klein, Bridging the gap: Automatic verified abstraction of c, in *Int. Conf. Interactive Theorem Proving*, 2012, pp. 99–115.
11. G. Klein *et al.*, seL4: Formal verification of an OS kernel, in *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles*, 2009, pp. 207–220.
12. G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski and G. Heiser, Comprehensive formal verification of an OS microkernel, *ACM Trans. Comput. Syst.* **32**(1) (2014) 1–70.
13. X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister and C. Ferdinand, Compcert-a formally verified optimizing compiler, in *8th European Congress ERTS 2016: Embedded Real Time Software and Systems*, 2016, https://hal.archives-ouvertes.fr/hal-01238879.
14. D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy and S. Blazy, CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler, in *ERTS2 2018-9th European Congress Embedded Real-Time Software and Systems*, 2018, pp. 1–9.
15. R. Krebbers and F. Wiedijk, A typed C11 semantics for interactive theorem proving, in *Proc. 2015 Conf. Certified Programs and Proofs*, 2015, pp. 15–27, https://doi.org/10.1145/2676724.2693571.
16. J. Biendarra, J. C. Blanchette, M. Desharnais, L. Panny, A. Popescu and D. Traytel, Defining (Co) datatypes and primitively (Co) recursive functions in Isabelle/HOL (2020).
17. R. Kaiser, C++ 11 smart pointer: shared_ptr, unique_ptr und weak_ptr, in *C++ mit Visual Studio 2017 und Windows Forms-Anwendungen*, 2018, pp. 781–799.
18. A. Krauss, Defining recursive functions in Isabelle/HOL (2008).
19. T. Nipkow, Programming and proving in Isabelle/HOL Technical Report, University of Cambridge (2013).
20. A. Lochbihler, Fast machine words in Isabelle/HOL, in *Int. Conf. Interactive Theorem Proving*, 2018, pp. 388–410.
21. F. Haftmann and T. Nipkow, Code generation via higher-order rewrite systems, in *Int. Symp. Functional and Logic Programming*, 2010, pp. 103–117.
22. M. Pierre and F. Burgos, Formalization of sorting algorithms in Isabelle/HOL (2019), pp. 1–45, https://matryoshka-project.github.io/pubs/fernandez_burgos_bsc_thesis.pdf.