applied sciences

MDPI

Article

Type Inference for Isabelle2Cpp

Dongchen Jiang *, Yongkang Mao and Chenxi Fu *

School of Information Science and Technology, Beijing Forestry University, Beijing 100083, China; jasonme@bjfu.edu.cn

* Correspondence: jiangdongchen@bjfu.edu.cn (D.J.); steve7800@outlook.com (C.F.)

Abstract

Isabelle2Cpp is a code generation framework that supports the automatic generation of C++ code from Isabelle/HOL specifications. However, when type information in an Isabelle/HOL specification is incomplete, Isabelle2Cpp may not be able to complete code generation automatically. To address this issue, this paper presents a type system for Isabelle2Cpp that performs type inference and type unification for expressions in its intermediate representation. The system introduces new type inference rules and unification algorithms to enhance the Isabelle2Cpp framework. By incorporating this type system, Isabelle2Cpp provides more comprehensive type information for expressions, leading to more complete and accurate C++ code generation.

Keywords: Isabelle2Cpp; type inference; type system; code generation

1. Introduction

Isabelle/HOL is a general interactive theorem-proving assistant that provides a formal environment for writing specifications and proving properties [1]. It has been widely used for the mechanical proof of mathematical theorems and for the verification of the correctness of algorithms, software, and complex systems. Users can define datatypes and functions and prove relevant properties.

To obtain executable code automatically from specifications, Isabelle2Cpp has been proposed to generate C++ implementations from Isabelle/HOL specifications [2]. This automatic conversion separates C++ code generation from the property verification of Isabelle/HOL specifications and combines the verification convenience of Isabelle/HOL with the execution efficiency of C++. Since the correctness of Isabelle/HOL specifications can be guaranteed through interactive proving, the correctness of the generated C++ code can also be preserved.

However, Isabelle/HOL and C++ have different type inference systems: when type information in an Isabelle/HOL specification is incomplete, Isabelle/HOL infers all missing types, whereas some variable types in the generated C++ code may not be correctly inferred using the auto-deltype-based type reasoning mechanism. Manual completion of missing types is still required in partially generated C++ code.

The purpose of this paper is to introduce an independent type system for the Isabelle2Cpp code generation framework. This system can infer expression types for specifications at all levels of granularity and perform type instantiation to support C++ code generation. It introduces new type inference rules and unification algorithms to enhance the Isabelle2Cpp framework. By incorporating this type system, Isabelle2Cpp provides more comprehensive type information for expressions and can automatically

Academic Editor: Andrea Prati

Received: 12 February 2026
Revised: 10 April 2026
Accepted: 12 April 2026
Published: 14 April 2026

generate corresponding C++ code, even when some type information is missing from the Isabelle/HOL specification.

This paper is organized as follows. Section 2 reviews related works. Section 3 describes the Isabelle2Cpp framework and analyzes problems in type inference. Section 4 provides the type inference rules of Isabelle2Cpp, and Section 5 describes their algorithmic implementation. Experiments are presented in Section 6, and Section 7 concludes this paper.

2. Related Works

Type systems play an important role in the design of programming languages and in type-based program analysis. In programming, a type system is a logical system consisting of a set of rules that assigns a type to each term (e.g., a variable, expression, or function) [3]. It is typically specified as part of a programming language. Type inference allows programmers to omit type annotations while still enabling type checking and ensuring that programs are well typed [4].

One of the earliest well-known type systems is System F, an extension of the lambda calculus with polymorphic types [5]. It was independently discovered by Jean-Yves Girard and John C. Reynolds in 1971 and named System F by Reynolds in 1972. A key feature of System F is the introduction of type abstraction and type application, which makes types first-class entities; i.e., they can be passed as parameters and returned as values of functions. This feature enables System F to support higher-order polymorphism.

The Hindley–Milner (HM) type system is another well-known type system, originally described by Roger Hindley [6] and later rediscovered by Robin Milner [7]. The main difference between the HM type system and System F lies in how polymorphism is expressed: the HM type system achieves local polymorphism through type inference over function parameters and return values. In contrast, System F supports global polymorphism, giving it greater expressive power and enabling it to represent more complex polymorphic patterns.

The HM system was first implemented as part of the ML programming language. A notable feature of the HM type system is its ability to infer the most general type of a given program without requiring type annotations or other hints from the programmer. Milner's Algorithm W is an effective type inference method that has been successfully applied to large codebases. In addition, Algorithm M is a top-down type inference algorithm for ML that detects type errors earlier.

Helen Soosaipillai proposed a method for type checking SML programs by combining ideas from interpretive reasoning [8]. Specifically, the type checker generates interpretations from existing types, thereby improving the efficiency of type checking. Bastiaan J. Heeren, Jurriaan Hage, and others described an approach that generates a set of type constraints for expressions [9]. These constraints are typically generated locally, allowing the framework to be applied to more complex type inference problems. Beyond the HM type system, Assaf J. Kfoury and Joe B. Wells defined an intersection type system with primary types and strongly normalizable λ-term types [10]. This system can handle complex cross-type inference and ensure the uniqueness and consistency of type inference. Dimitrios Vytiniotis, Simon Peyton Jones, and others proposed a constraint-based local type inference method based on the OutsideIn(X) system, whose parameterization is similar to HM(X) over a given constraint domain X [11]. This constraint solver has been implemented as part of GHC 7. Dmitriy Traytel, Stefan Berghofer, and others extended a simple typed lambda calculus with Hindley–Milner polymorphism by adding coercive subtyping and proposed a type inference algorithm that determines where to insert such coercions to ensure correct typing [12]. The algorithm is reliable and complete and has been implemented in Isabelle.

In non-functional languages, Jens Palsberg and Michael Schwartzbach proposed a method for type inference in non-object-oriented programs that include inheritance, as-

signments, and late binding [13]. The method guarantees that all messages are understood, annotates programs with type information, supports polymorphic methods, and serves as a basis for optimizing compilers. Motivated by statistical analysis of type inference in ML codebases, Benjamin C. Pierce and David N. Turner explored two partial type inference methods for languages that combine subtyping and imperative polymorphism [14]. For type inference in stripped binary files, Ligeng Chen, Zhongling He, and others proposed the CATI algorithm [15], which captures the context of function calls during execution and infers the types of function arguments and return values. To address the limitations of traditional methods, Vincent J. Hellendoorn, Christian Bird, and others proposed DeepTyper, a deep learning-based type inference system [16]. The core idea of DeepTyper is to infer types of code by learning structural and semantic information from code fragments rather than relying on static rules and type systems.

The type system of Isabelle/HOL follows the standard HM type system, which differs from the auto-deltype-based type reasoning mechanism in C++. This difference may lead to inconsistencies between the inferred types of some variables in the C++ code generated by Isabelle2Cpp and the corresponding terms in the Isabelle/HOL specifications. Because the Isabelle/HOL type system cannot be applied directly to infer types in the generated C++ code, it is necessary to provide a corresponding type system for Isabelle2Cpp.

3. Framework of Isabelle2Cpp

The architecture of Isabelle2Cpp is shown in Figure 1. For a given specification in Isabelle/HOL, a standard abstract syntax tree (AST) is first constructed through syntax parsing; it contains all semantic information of the specification. Next, all information stored in the AST, e.g., specification names, type variables, datatypes and their corresponding construction rules, and equations defining functions, is used in the core part of C++ code generation, and all conceptional generation is performed in the conversion phase. Finally, the synthesis phase produces concrete C++ files by adding the corresponding header files so that the program can be compiled and executed directly.

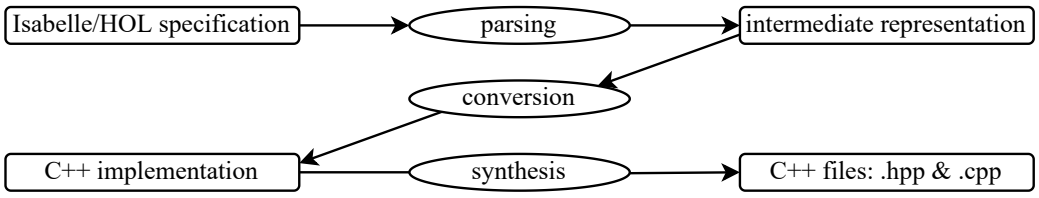

Figure 1. Architecture of Isabelle2Cpp.

The Isabelle2Cpp framework supports two conversion approaches: definition-based conversion and rule-based conversion.

The definition-based method converts Isabelle/HOL datatypes into C++ classes or class templates, where each construction rule of a datatype is represented as a nested C++ class. It also converts Isabelle/HOL functions into sequences of semantically equivalent statements in the generated C++ functions or function templates, where each equation of an Isabelle/HOL function is translated into an if-statement block (referred to as a conditional statement module, or CSM). During function conversion, the pattern of an equation is translated into an if-statement condition and the associated variable declarations, while the expression of the same equation is translated into a final return statement and several auxiliary statements. The correctness of the definition-based conversion is guaranteed by the following facts:

1. The structures are carefully designed to be isomorphic: not only do the structures of the C++ code match those of the specifications, but the elementary entities of C++ are also in one-to-one correspondence with those of the Isabelle/HOL specifications.
2. The basic statements in the CSM of the generated C++ code are semantically equivalent to the basic operations of the corresponding Isabelle/HOL specification.

Rule-based conversion improves the efficiency of the generated C++ code. Rules for datatype and operation conversion are manually provided by the user; thus, existing or predefined C++ types and relevant operations can be directly utilized during code generation. To guarantee correctness, the corresponding entities (datatypes/types or functions) in a conversion rule must express the same mathematical concepts in both languages. In general, rule-based conversion is more flexible and produces more efficient C++ code, but its correctness depends on the correctness of the user-defined conversion rules.

Problems

Isabelle2Cpp automatically generates executable C++ code if the Isabelle/HOL specification contains all necessary information, e.g., type names, type variables, construction rules for type definitions, function names, function types, and equations for function definitions (see Figure 2). However, because Isabelle/HOL supports type inference, some type information may be omitted in specifications without causing errors. This omission leads to missing type information in the AST and prevents the generated C++ code from being compiled directly.

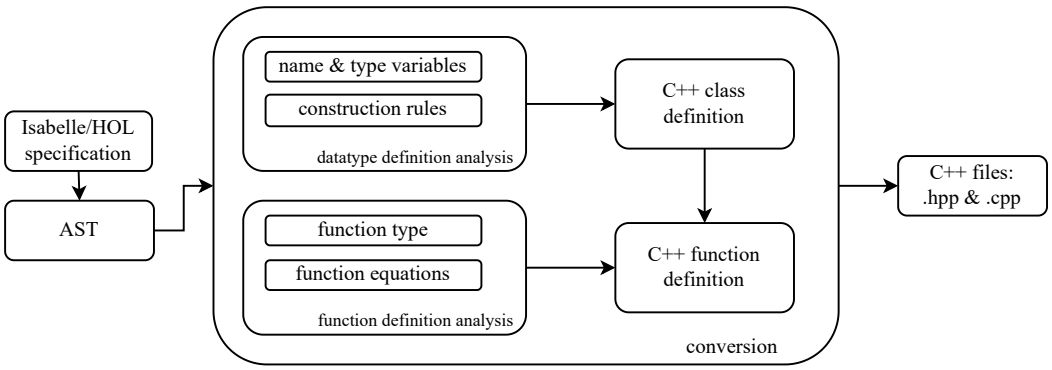

Figure 2. Conversion process of Isabelle2Cpp.

To address the issue of missing type information in generated C++ code, Jiang and Xu used the **decltype** specifier to inspect the type of an entity. To determine the types of newly declared variables, the framework first checks whether type information is available in the AST. If no type information is found, unlabeled variables are declared using **auto**; the specific type of an **auto** variable is then inferred using **decltype** in C++, based on the existing type information of the relevant expression.

```
fun test :: "α list => nat" where
  "test Nil = 0" |
  "test (Cons x xs) = length (If ((length xs) = 0) Nil xs) + 1"
```

```
std::uint64_t test(const std::list<T1> &arg1) {
    ...
    // test (Cons x xs) = length (If ((length xs) = 0) Nil xs) + 1
    if (!arg1.empty()) {
        auto xs = decltype(arg1){std::next(arg1.begin()), arg1.end()};
        UNKNOWN_TYPE temp0;
        if (xs.size() == 0) {
            temp0 = {};
        } else {
            temp0 = xs;
        }
        return temp0.size() + 1;
    } else { // auto-generated for -Wreturn-type
        std::abort();
    }
}
```

Take the Isabelle/HOL function *test* as an example. Without type inference, Isabelle2Cpp cannot infer that *xs* is of type α *list* and thus specifies *xs* as an auto variable in C++. Because the type of *xs* is the same as that of $(Cons\ x\ xs)$, which corresponds to the type of the first parameter $arg1$ in the generated C++ function, the type of *xs* can be obtained using **decltype**; i.e., $decltype(arg1)\{std :: next(arg1.begin()),\ arg1.end()\}$.

decltype is useful for inspecting the types of specific expressions at the C++ code level, e.g., lambda-related types or types that depend on template parameters. However, obtaining type information at the C++ code level still cannot resolve all type-related problems in code generation.

First, **decltype** cannot address type unification. For example, in the specification of *test*, If can be regarded as a function with type $bool \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha$. Isabelle2Cpp converts the If expression into an if-statement block with an additional temporary variable $temp0$. According to the datatype of the If expression in Isabelle/HOL, the types of its second and third parameters, as well as the return type of the entire expression, are identical. However, due to the absence of a type unification mechanism, Isabelle2Cpp cannot assert that the type of the second parameter *Nil* is the same as that of the third parameter *xs* based on the type of the If expression. Although the type of *xs* can be specified using **decltype** during compilation, the type of $temp0$ cannot be inferred and is therefore denoted as UNKNOWN_TYPE, which must be specified manually afterwards.

Second, in an Isabelle/HOL specification, a lambda expression cannot only be used in a function call but also be passed as a parameter to a higher-order function. In the latter case, Isabelle2Cpp uses the std:: function class template to represent the types of function parameters, which should be instantiated with the type of the corresponding lambda expression. However, C++ lambda expressions are implemented as anonymous structures, which leads to inconsistency when the C++ lambda expression is used as a parameter of a higher-order function, as required by the original Isabelle/HOL specification.

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

Consider the higher-order function *map* as an example. Isabelle2Cpp generates a function template for *map* in C++, where the type of its first parameter is std::function<R(T)>. If an **auto** variable f is declared to receive a lambda expression in the main function, its type is deduced as a unique, unnamed closure type [] (auto x) $\rightarrow$ auto, which is inconsistent with the parameter type std::function<R(T)>. However, if the specific type of a lambda

expression can be determined prior to code generation, then the type of the std::function class template in a higher-order function can be instantiated automatically. For example, if the parameter type and return type of a lambda expression are both *nat*, then a std::function<int<int> >type variable can be used to receive the lambda expression.

```
template<typename T, typename R>
std::vector<R> map(std::function<R(T)> f, std::vector<T> xs) {
    auto impl = [&]() -> R {
        //map f [] = []
        if (xs.empty()) {
            return R();
        }
        //map f (x : xs) = (f x) : (map f xs)
        auto x = xs.front();
        xs.erase(xs.begin());
        return R({f(x)}) + map(f, xs);
    };
    static std::map<std::tuple<std::function<R(T)>, std::vector<T>>, R> cache;
    auto args = std::make_tuple(f, xs);
    auto it = cache.find(args);
    return it != cache.end() ? it->second : (cache.emplace(std::move(args),
        impl()).first->second);
}

int main() {
    auto f = [](auto x) -> auto { return x * x; };
    auto xs = std::vector<std::uint64_t>({1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
    auto ys = map(f, xs);
    for (auto y : ys) {
        std::cout << y << std::endl;
    }
}
```

The example shows that type inference at the C++ code level alone cannot eliminate the problems caused by missing type information in Isabelle/HOL specifications. Therefore, it is necessary to provide a type system for Isabelle2Cpp.

4. Type System in Isabelle2Cpp

4.1. Function Specifications in Isabelle/HOL

In Isabelle/HOL, a function is defined by specifying its name, type, and the relationship between its inputs and output. The specification follows the syntax [17]:

$$\begin{aligned}&\textbf{fun } f :: {''}t_1 \Rightarrow ... \Rightarrow t_{\mathrm{N}} \Rightarrow t''_{\mathrm{R}} \textbf{ where}\\ &{''}pattern_1 = expression''_1\\ &\vdots\\ &{''}pattern_{\mathrm{n}} = expression''_{\mathrm{n}}\end{aligned}$$

Here, f denotes the function name; $t_1, \ldots, t_{\mathrm{N}}$ are the parameter types; t_{R} is the return type; and each equation $pattern_{\mathrm{i}} = expression_{\mathrm{i}}$ $(\mathrm{i} = 1, \ldots, n)$ defines f using pattern matching. On the left-hand side of each equation, a pattern is typically represented by a datatype constructor with associated pattern parameters; on the right-hand side, the corresponding expression defines the function.

During the conversion process of Isabelle2Cpp, all necessary information for code generation is parsed and stored in the AST. Accordingly, the structure of the AST is modified to include type nodes for all pattern parameters and expressions, including their corresponding sub-expressions (see Figure 3).

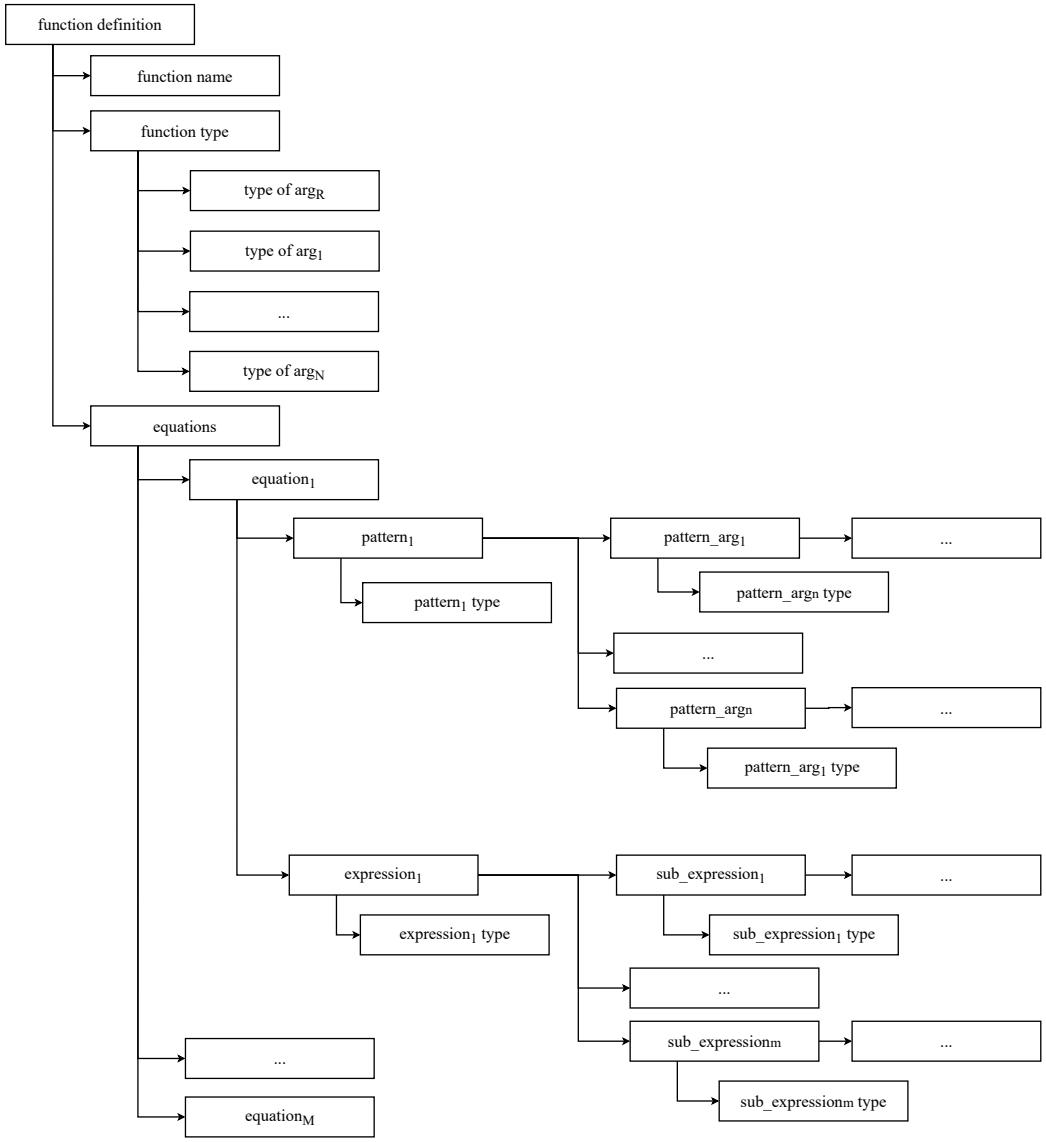

Figure 3. An abstract syntax tree with type information variable type.

4.2. Expressions and Type Constructions in Isabelle2Cpp

To define the type system of Isabelle2Cpp, expressions in the ST are formally defined as follows:

$$\begin{aligned} e \ := x \mid (C\ e_1\ e_2\ \cdots\ e_n) \mid (if\ e\ then\ e'\ else\ e'') \mid (\lambda\ x_1\ \cdots\ x_n.\ e) \mid \\ (case\ e\ of\ pattern_1\ =>\ e_1 \mid pattern_m\ =>\ e_m) \mid \\ (let\ x\ =\ e\ in\ e') \mid [e_1\ \cdots\ e_n] \mid \{e_1\ \cdots\ e_n\} \mid \cdots \end{aligned}$$

According to this definition, an expression in the AST can be a constant (e.g., $1, 2, True, False$), a variable, an application expression, an if-expression, a lambda expression, a case expression, a let-in expression, a list expression, or a set expression. All these forms are commonly used in Isabelle/HOL specifications. Function application expressions and value construction expressions in Isabelle/HOL are both treated as application expressions in this definition. This is because an n-ary function and an n-ary value constructor share

similar characteristics: once n values of appropriate types are provided, a new value is obtained via function application or value construction. In this definition, list and set expressions can be regarded as special cases of function application expressions. They are specified explicitly because the rule-based conversion of Isabelle2Cpp supports direct conversion for the corresponding datatypes.

To provide type information for expressions, type expressions are defined as follows:

$$\tau := \alpha \mid T \mid (\tau) \mid \tau_1 \Rightarrow \tau_2 \mid (\tau_1,\ \tau_2) \mid \tau_1 \ldots \tau_n\ D \mid \alpha\ list \mid \alpha\ set\ \ldots$$

According to this definition, a type expression can be a type variable $(\alpha, \beta, etc.)$, a primitive type $(bool, nat, etc.)$, a function type, a tuple type, a new type constructed by applying an abstract type constructor to existing types, a list type, or a set type. For example, the constant "True" has type bool, which is a primitive type, and "$\lambda\ x.\ x + 1$" has type nat $\Rightarrow$ nat.

As polymorphic types are supported by both Isabelle/HOL and Isabelle2Cpp, a type variable α denotes a set of types. To simplify the description of type analysis, the set of type variables $Var(\tau)$ for a given type τ is inductively defined as follows.

Definition 1. *For a given type τ:*

- *If τ is a type variable α, then $Var(\tau) = \{\alpha\}$;*
- *If τ is a primitive type T, then $Var(\tau) = \emptyset$;*
- *If $\tau = (\tau_1)$, then $Var(\tau) = Var(\tau_1)$;*
- *If $\tau = \tau_1 \Rightarrow \tau_2$, then $Var(\tau) = Var(\tau_1) \cup Var(\tau_2)$;*
- *If $\tau = (\tau_1, \tau_2)$, then $Var(\tau) = Var(\tau_1) \cup Var(\tau_2)$;*
- *If $\tau = \tau_1 \ldots \tau_n\ D$, then $Var(\tau) = Var(\tau_1) \cup \cdots \cup Var(\tau_n)$;*
- *If $\tau = \alpha\ list$, then $Var(\tau) = \{\alpha\}$;*
- *If $\tau = \alpha\ set$, then $Var(\tau) = \{\alpha\}$.*

Here, $Var(\tau)$ denotes the set of all type variables occurring in the type τ.

4.3. Rules of Type Inference in Isabelle2Cpp

In Isabelle2Cpp, if e is an expression and τ is a type expression, the typing judgment $e : \tau$ asserts that e has type τ. A type context Γ maps variables to types and can be represented as $\{v_1 : \tau_1, \ldots,\ v_n : \tau_n\}$. A typing formula is a formal statement of the form $\Gamma \vdash e : \tau$, where Γ is a type context, e is an expression, and τ is a type. A derivable typing is one that can be inferred from a consistent type context via the type system. For example, the typing formula $\{c : nat, (\lambda n.n + 1) : nat \Rightarrow nat\} \vdash (\lambda n.n + 1)c : nat$ means that $(\lambda n.n + 1)c$ has type nat, as inferred from the given type context.

The inference rules of the Isabelle2Cpp type system are divided into three categories: top-down rules, bottom-up rules, and unification rules. In Isabelle/HOL, functions are defined using pattern matching. Accordingly, the type system first obtains general type information for function parameters from the specification and then infers the specific types of pattern variables based on the corresponding abstract type constructors. Top-down rules propagate type information from expressions to their sub-expressions. Specifically, the top-down rules for abstract type constructors are given by **App-TD**, **List-TD**, and **Set-TD**; these rules also support type instantiation for all sub-expressions. Additionally, **Let-TD** and **Case-TD** handle let-in and case expressions.

For example, consider the binary search specification *bs* in Section 6.1. From the specification, the parameter [y] has type *nat list*, and the type of the variable y can then be inferred as *nat* using **List-TD**.

$$\frac{\Gamma \vdash [e_1, \cdots e_n] : \tau\ list}{\Gamma \vdash e_1 : \tau, \cdots e_n : \tau} \quad [\textbf{List-TD}]$$

$$\frac{\Gamma \vdash \{e_1, \cdots e_n\} : \tau\ set}{\Gamma \vdash e_1 : \tau, \cdots e_n : \tau} \quad [\textbf{Set-TD}]$$

$$\frac{\Gamma \vdash C\ e_1\ \cdots e_n : \tau_r \quad \sum(C) = \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau_r}{\Gamma \vdash\ e_1 : \tau_1, \cdots, e_n : \tau_n} \quad [\textbf{App-TD}]$$

$$\frac{\Gamma \vdash \textbf{let}\ x = e_0\ \textbf{in}\ e_1 : \tau}{\Gamma \vdash e_1 : \tau} \quad [\textbf{Let-TD}]$$

$$\frac{\Gamma \vdash (\textbf{Case}\ e\ \textbf{of}\ pattern_1 \Rightarrow e_1\ \cdots\ pattern_m \Rightarrow e_m) : \tau}{\Gamma \vdash e_1 : \tau \cdots, e_m : \tau} \quad [\textbf{Case-TD}]$$

A type solver Σ records the types of functions or datatype constructors. For example, $\Sigma(Cons) = \alpha \Rightarrow \alpha\ list \Rightarrow \alpha\ list$ indicates that the constructor *Cons* has type $\alpha \Rightarrow \alpha\ list \Rightarrow \alpha\ list$, constructing a new list by adding an element of type α to an existing list of type $\alpha\ list$. Similarly, $\Sigma(length) = \alpha\ list \Rightarrow nat$ indicates that the function *length* has this type.

Bottom-up rules derive the type of an expression from its immediate sub-expressions:

$$\frac{e : \sigma \in \Gamma}{\Gamma \vdash e : \sigma} \quad [\textbf{Exp-BU}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1, \cdots\ e_n : \tau_n \quad \sum(C) = \tau_1 \cdots \tau_n \Rightarrow \tau_r}{\Gamma \vdash\ C\ e_1\ \cdots\ e_n : \tau_r} \quad [\textbf{App-BU}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \textbf{let}\ x = e_0\ \textbf{in}\ e_1 : \tau} \quad [\textbf{Let-BU}]$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e_1 : \tau' \cdots \Gamma \vdash e_m : \tau'}{\Gamma \vdash (\textbf{Case}\ e\ \textbf{of}\ pattern_1 \Rightarrow e_1 \cdots\ pattern_m \Rightarrow e_m) : \tau',} \quad [\textbf{Case-BU}]$$

$$\Gamma \vdash pattern_1 : \tau \cdots pattern_m : \tau$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau'}{\Gamma \vdash \lambda x_1, \ldots x_n.e : \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau'} \quad [\textbf{Abs-BU}]$$

$$\frac{\Gamma \vdash e_1 : \tau, \cdots, e_n : \tau}{\Gamma \vdash [e_1, \cdots, e_n] : \tau\ list} \quad [\textbf{List-BU}]$$

$$\frac{\Gamma \vdash e_1 : \tau, \cdots, e_n : \tau}{\Gamma \vdash \{e_1, \cdots, e_n\} : \tau\ set} \quad [\textbf{Set-BU}]$$

The rule **Exp-BU** is atomic: a typing can be inferred directly from the context (e.g., $\{..., e : \sigma, ...\} \vdash e : \sigma$). **App-BU** handles type inference for value construction and function application. Because Isabelle/HOL supports currying, the following rule captures partial application as a specialization of **App-BU**:

$$\frac{\sum(F) = \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau_r \quad \Gamma \vdash e_1 : \tau_1, \cdots e_i : \tau_i \quad 1 \leq i < n}{\Gamma \vdash F\ e_1\ \cdots e_i : \tau_{i+1} \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau_r} \quad [\textbf{Currying}]$$

An expression's type may be inferred via top-down or bottom-up rules, and different rules may yield different types. A type unification mechanism resolves such discrepancies by finding substitutions for type variables that make the type expressions identical.

Unification identifies substitutions that equate two type expressions. Determining which variables to substitute requires the binary abstract–concrete relation $\succeq$ (and $\succ$), defined inductively as follows.

Definition 2. *The abstract–concrete relation* $\succeq$*:*

- *If* $\forall \alpha,\ \beta \in TV$*, then* $\alpha \succeq \beta,\ \beta \succeq \alpha$*;*
- *If* $\forall \alpha \in TV,\ T \in BT$*, then* $\alpha \succ T$*;*

- *If* $\tau \succeq \sigma$, *then* $(\tau) \succeq (\sigma)$;
- *If* $\tau_1 \succeq \sigma_1$, *then* $(\tau_1, \tau_2) \succeq (\sigma_1, \sigma_2)$;
- *If* $\tau_1 \succeq \sigma_1, \ldots \tau_n \succeq \sigma_n$, *then* $\tau_1 \ldots \tau_n\ D \succeq \sigma_1 \ldots \sigma_n\ D$;
- *If* $\tau_1 \succeq \sigma_1, \ldots \tau_n \succeq \sigma_n$, *then* $\tau_1 \Rightarrow \cdots \Rightarrow \tau_n \succeq \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n$.

Here, BT denotes the set of primitive types (e.g., $nat, bool$), TV is the set of type variables (e.g., α,β); and D is an n-ary abstract type constructor with type parameters τ_i. The relation $\succeq$ is replaced with $\succ$ if any premise uses $\succ$.

The "abstract–concrete" relation differs from subtyping. Given two type variables α and β, neither is inherently more abstract or more concrete; both $\alpha \succeq \beta$ and $\beta \succeq \alpha$ hold. Consequently, $\alpha \Rightarrow \alpha \succeq \alpha \Rightarrow \beta$ and $\alpha \Rightarrow \beta \succeq \alpha \Rightarrow \alpha$ both hold. In contrast, under subtyping, $\alpha \Rightarrow \alpha$ would be a subtype of $\alpha \Rightarrow \beta$, which differs from the "abstract–concrete" relation.

This relation supports top-down analysis: relationships between type expressions propagate to sub-expressions and variables. Within a given type context, once the "abstract–concrete" relation between two type expressions is determined, it can be reduced to relations over simpler types according to the type structure. This reduction, denoted by $\Longrightarrow$, is defined as follows.

Definition 3. *Reduction:*

- $(\tau) \succeq (\sigma) \Longrightarrow \tau \succeq \sigma$;
- $(\tau_1, \tau_2) \succeq (\sigma_1, \sigma_2) \Longrightarrow \tau_1 \succeq \sigma_1, \tau_2 \succeq \sigma_2$;
- $\tau_1 \ldots \tau_n\ D \succeq \sigma_1 \ldots \sigma_n\ D \Longrightarrow \tau_1 \succeq \sigma_1, \ldots \tau_n \succeq \sigma_n$;
- $\tau_1 \Rightarrow \cdots \Rightarrow \tau_n \succeq \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Longrightarrow \tau_1 \succeq \sigma_1, \ldots \tau_n \succeq \sigma_n$.

Reduction is an inference process over the concrete–abstract relation. For a relation $\tau \succeq \sigma$, reduction can be applied repeatedly until an irreducible form is obtained. It can be shown inductively that, in the final reduction, the left-hand side must be a type variable. That is, if $\tau \succeq \sigma$ reduces to $\alpha \succeq \delta$, where α is a type variable, then $\alpha \succeq \delta$ is irreducible. We write $\tau \succeq \sigma \overset{*}{\Longrightarrow} \alpha \succeq \delta$ for the full reduction sequence and define the corresponding type substitution as $\alpha := \delta$. Because reduction may branch, the final result can be represented as a set of substitutions. In this paper, $\mathcal{S}$ denotes a substitution set, i.e., $\mathcal{S} = \{\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n\}$.

Based on the abstract–concrete relations and reduction, type unification is defined as

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e : \sigma \quad \tau \succeq \sigma \quad \tau \succeq \sigma \overset{*}{\Longrightarrow} \mathcal{S}}{\Gamma = \mathcal{S}\Gamma} \ [\textbf{Uni}]$$

Here, $\mathcal{S}\Gamma$ denotes the application of the substitutions in $\mathcal{S}$ to Γ. The unification rule **Uni** can be specialized to application and lambda expressions as follows:

$$\frac{\Gamma \vdash F\ e_1 \cdots e_n : \tau_r \quad \sum(F) = \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau_r \quad \Gamma \vdash e_i : \sigma_i \quad \Gamma \vdash e_i : \tau_i}{\sigma_i \succeq \tau_i \overset{*}{\Longrightarrow} \mathcal{S},\ \Gamma = \mathcal{S}\Gamma\ if\ \sigma_i \succeq \tau_i,\ \tau_i \succeq \sigma_i \overset{*}{\Longrightarrow} \mathcal{S},\ \Gamma = \mathcal{S}\Gamma\ if\ \tau_i \succeq \sigma_i} \ [\textbf{Uni-App}]$$

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash F\ e_1 \cdots e_n : \tau_r \quad \sum(F) = \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \sigma_r \quad e_i = x_j \quad 1 \le i, j \le n}{\Gamma \vdash \lambda x_1 \ldots x_j \ldots x_n.\ F\ e_1 \cdots e_n : \tau_1 \Rightarrow \cdots \tau_{j-1} \Rightarrow \sigma_i \Rightarrow \tau_{j+1} \cdots \Rightarrow \tau_n \Rightarrow \tau_r} \ [\textbf{Uni-Abs}]$$

5. Type Inference of Isabelle2Cpp

The type inference system of Isabelle2Cpp consists of three modules (see Figure 4): the pattern parameter type extraction module, the bottom-up type inference module, and the top-down type completion module.

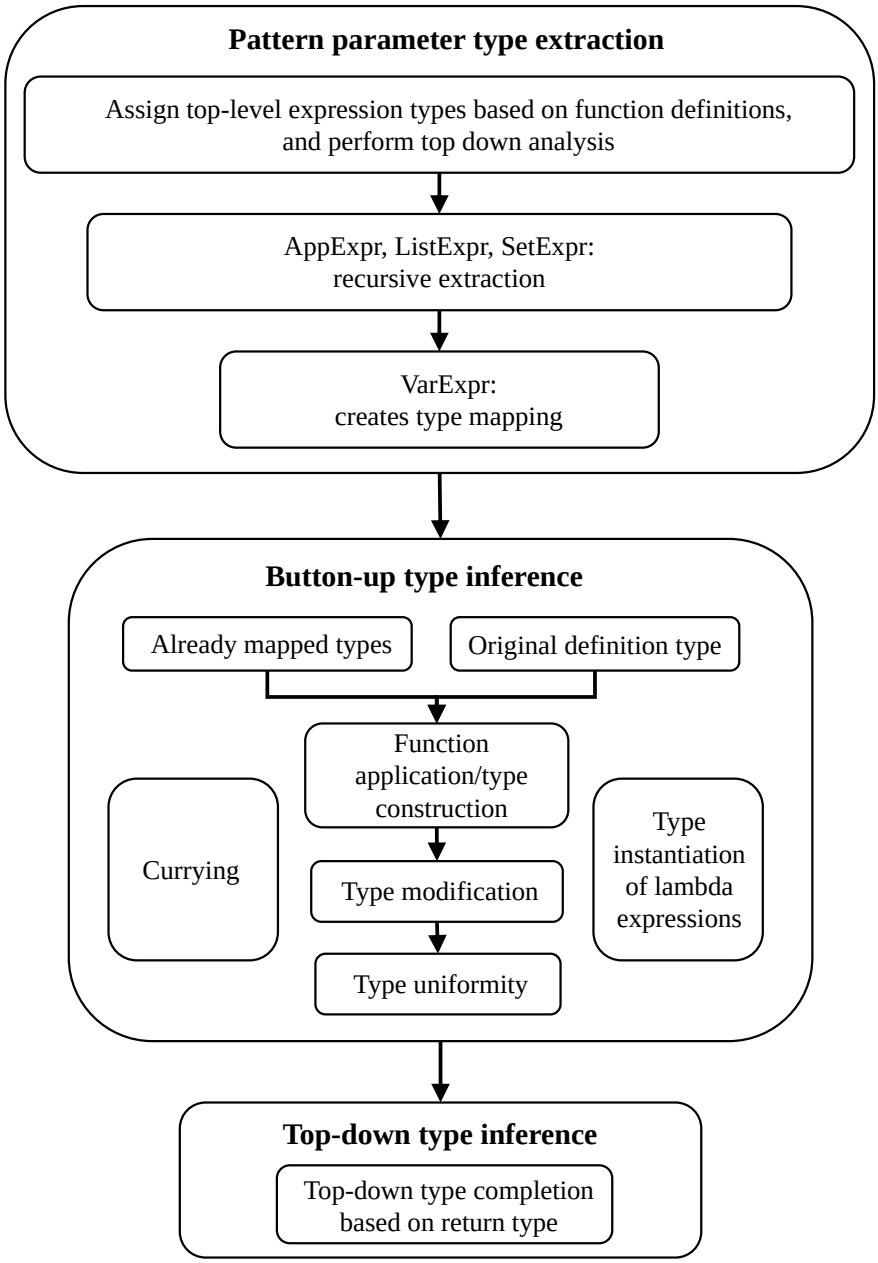

Figure 4. Modules of the type system.

5.1. Pattern Parameter Type Extraction Module

In an Isabelle/HOL function specification, an expression on the right-hand side of an equation may consist of existing functions, datatype constructors, and pattern parameters that appear on the left-hand side of the same equation. The types of existing functions and datatype constructors are defined in previously specified definitions and are recorded by the type solver Σ. Therefore, to infer the types of the expression and its sub-expressions on the right-hand side, it suffices to extract the type information of all pattern parameters on the left-hand side.

The pattern parameter type extraction module obtains the types of all pattern parameters. In a function specification, a pattern on the left-hand side of an equation consists of a datatype constructor and its associated parameters or sub-patterns. Because the type of the pattern is determined by the type of the corresponding function parameter, the types of its pattern parameters (or sub-patterns) can be inferred using top-down rules. The Algorithm 1 for pattern parameter type extraction is shown below.

Algorithm 1 $EX(\Gamma, e)$

if $e \in AppExpr$ **then**
  $\sum(e.constructor) = \tau_1 \Rightarrow \tau_2 \cdots \Rightarrow \tau_n \Rightarrow \tau_r$
  for $i = 1$ *to* $e.args\_num$ **do**
    $\Gamma = \Gamma \cup \{e.arg_i : \tau_i\}$
    $EX(\Gamma, e.arg_i)$
  end for
else if $e \in ListExpr$ **then**
  $\Gamma \vdash e : \tau\ list$
  for $i = 1$ *to* $e.exprs\_num$ **do**
    $\Gamma = \Gamma \cup \{e.expr_i : \tau\}$
    $EX(\Gamma, e.expr_i)$
  end for
else if $e \in SetExpr$ **then**
  $\Gamma \vdash e : \tau\ set$
  for $i = 1$ *to* $e.exprs\_num$ **do**
    $\Gamma = \Gamma \cup \{e.expr_i : \tau\}$
    $EX(\Gamma, e.expr_i)$
  end for
else if $e \in VarExpr$ **then**
  return
else if $e \in ConstExpr$ **then**
  return
else
  $\Gamma = \Gamma \cup \{e : \bot\}$
end if

In the pattern parameter type extraction algorithm EX, the input Γ is the environment containing all type information for the function parameters, and e is the pattern expression to be processed. In a function specification, a pattern expression e is either a variable expression ($VarExpr$) or a value construction expression ($ConstExpr$ or $AppExpr$). If the current pattern expression is $VarExpr$ or $ConstExpr$, it has no sub-expressions, and the algorithm terminates for the current branch. If it is $AppExpr$, the algorithm obtains the type of the constructor $e.constructor$ from Σ and assigns each pattern parameter $e.arg_i$ to the corresponding parameter type of $e.constructor$.

As special cases of $AppExpr$, $ListExpr$ and $SetExpr$ may also appear in pattern expressions. In such cases, the overall type of the expression (i.e., $\tau\ list$ or $\tau\ set$) must already have been derived and stored in the type environment Γ. The algorithm first retrieves this overall type from Γ and then applies the **List-TD** or **Set-TD** rule to infer the types of its elements. For other kinds of expressions, the undefined type $\bot$ is assigned.

The algorithm proceeds recursively until the types of all pattern expressions and variables have been inferred. After processing, all variables and their corresponding type information are stored in Γ, which is then incorporated into the enriched AST.

5.2. Bottom-Up Type Inference Module

The bottom-up type inference module infers the types of all expressions on the right-hand side of function equations. It consists of three components: type modification, type unification, and bottom-up type inference.

5.2.1. Type Modification

Type modification eliminates conflicts among type variables used by different functions and type constructors. Type variables serve as parameters for abstract type construction in specifications. The symbols used for type variables in a given specification are interchangeable and serve only to distinguish different type variables within that specification. If type variables are treated as fixed symbols, conflicts may arise during type inference. To address this issue, type modification resolves conflicts caused by different functions using identical type variable symbols.

Type modification is applied when an expression is assigned its originally defined type. In such cases, the names of the type variables in the expression's type are modified to distinguish them from those in the original definition. Specifically, a counting scheme is used to ensure uniqueness. For example, the original type of the map function is $(\beta \Rightarrow \gamma) \Rightarrow \beta\ list \Rightarrow \gamma\ list$, Nil, and the original type of the constructor Nil is $\alpha\ list$. During assignment, the type of map is modified to $(\beta\#\_ \Rightarrow \gamma\#\_) \Rightarrow \beta\#\_\ list \Rightarrow \gamma\#\_\ list$, and the type of Nil is modified to $\alpha\#\_\ list$. Here, "_" denotes a counter value that is incremented after each modification to avoid symbol conflicts arising from type assignment.

Note that a function's type may also originate from a left-hand pattern. In this case, a variable with a function type appearing on the left-hand side is treated as having the actual type, and no modification is applied.

5.2.2. Type Unification

Type unification is the process of finding substitutions that make the inferred types of an expression consistent. During type inference, Γ denotes the environment that stores the inferred type information for expressions. Because type information may be obtained from different sources using different inference rules, an expression may be assigned different types in Γ. In such cases, these types must be unified. In this paper, type unification is applied to function application expressions and function abstraction expressions, i.e., $AppExpr$ and $LambdaExpr$. The unification Algorithm 2 is presented below.

For a function application expression, the types of its parameters can be obtained from two sources: (i) bottom-up inference applied to the parameter itself, and (ii) the function (or constructor) type, determined by the parameter's position in the application. Consequently, a parameter expression may have two distinct type expressions. When this happens, the algorithm performs one or more reductions based on the abstract–concrete relation $\succeq$ between the two type, and applies the resulting substitution set $\mathcal{S}_i$ to update the corresponding type variables in Γ. After substitution, the types of the function application expression and its parameters are unified.

For a function abstraction expression, i.e., a lambda expression, the symbol λ introduces a set of parameters that bind the lambda body. When the body is a function application (or a value construction), a λ-parameter appearing in that expression (or the value construction) may also have two sources of type information: the type introduced by the λ-parameter itself and the type inferred from the function application. Therefore, the types of the parameters and of the entire lambda expression must be unified.

For other types of expressions, where no inconsistent type expressions arise, type inference terminates for those branches. Ultimately, the function parameters and their corresponding type information are stored in Γ and recorded in the modified abstract syntax tree (AST).

Consider the following function, which computes the length of a linked list:

```
fun test :: α list ⇒ nat where
  test Nil = 0
  test (Cons x xs) = length (If ((length xs) = 0) Nil xs) + 1
```

In this specification, If has type $bool \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha$, xs has type $\alpha\ list$, and Nil has type $\alpha\ list$. However, because these types originate from different sources, they cannot initially share the same type variable. According to their origins, xs inherits its type from the function parameter, i.e., $\alpha\ list$; the type of If is annotated as $bool \Rightarrow \alpha\#1 \Rightarrow \alpha\#1 \Rightarrow \alpha\#1$, and the type of Nil is annotated as $\alpha\#2\ list$.

Algorithm 2 $Unify(\Gamma, e)$

if $e \in AppExpr$ **then**
  $\sum(e.constructor) = \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau_r$
  for $i = 1$ *to* n **do**
    if $\Gamma \vdash e_{arg_i} : \tau_i$ *and* $\Gamma \vdash e_{arg_i} : \sigma_i$ **then**
      if $\sigma_i \succeq \tau_i$ **then**
        $\sigma_i \succeq \tau_i \overset{*}{\Longrightarrow} \mathcal{S}_i$
        $\Gamma = \mathcal{S}_i\Gamma$
      else
        continue
      end if
    end if
  end for
else if $e \in LambdaExpr$ **then**
  if $e.expr \in AppExpr$ **then**
    $\sum(e.expr.contructor) = \sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \sigma_r$
    $\Gamma,\ e.parameter_1 : \tau_1,\ \ldots\ e.parameter_n : \tau_n \vdash e.expr : \tau_r$
    for $1 \leq i,\ j \leq n$ **do**
      if $e_i = x_j$ **then**
        $\tau_j = \sigma_i$
        $\tau_e = \tau_1 \Rightarrow \cdots \Rightarrow \tau_{j-1} \Rightarrow \sigma_i \Rightarrow \tau_{j+1} \cdots \Rightarrow \tau_n \Rightarrow \tau_r$
      end if
    end for
  else
    return
  end if
else
  return
end if

The type unification process for this specification is illustrated in Figure 5. In the first step, the type of the first parameter of If requires no unification. However, the second parameter Nil has type $\alpha\#2\ list$, which is inconsistent with the expected type. Therefore, following standard function application rules, the substitution $\alpha\#1 := \alpha\#2\ list$ is introduced, yielding the instantiated function type $bool \Rightarrow \alpha\#2\ list \Rightarrow \alpha\#2\ list \Rightarrow \alpha\#2\ list$. In the second step, the third argument xs, with type $\alpha\ list$, is processed. Because α denotes a concrete type from the specification, the rule **Uni-App** yields the substitution $\alpha\#2 := \alpha$. Consequently, the types of the If function and its arguments are updated: the type of Nil becomes $\alpha\ list$, consistent with xs; and the type of If becomes $bool \Rightarrow \alpha\ list \Rightarrow \alpha\ list \Rightarrow \alpha\ list$, with result

type $\alpha\ list$, which is consistent with the type of the test function. Thus, type derivation and unification are completed.

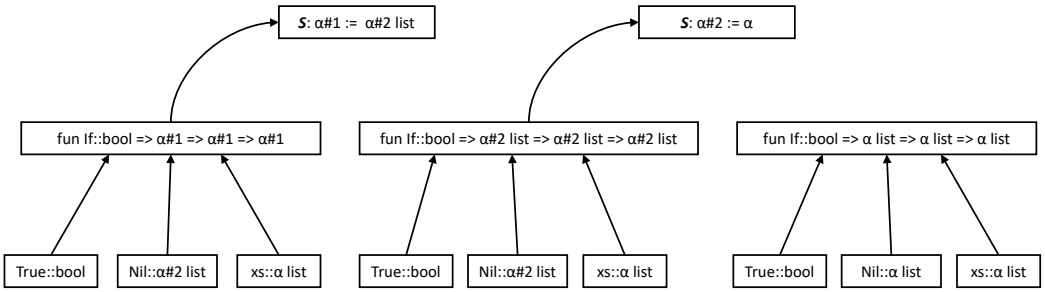

Figure 5. An example of type unification.

5.2.3. Bottom-Up Type Inference

The Isabelle2Cpp type system uses a bottom-up approach to infer the types of expressions on the right-hand side of function equations. In this process, the base expressions for type inference are constant expressions and variable expressions, i.e., $ConstExpr$ and $VarExpr$, whose types are obtained directly or via pattern parameter type extraction. For a compound expression, once the types of all its sub-expression are available, bottom-up inference rules are applied to determine its type. The bottom-up type inference Algorithm 3 is presented below.

Algorithm 3 $BU(\Gamma,\ e)$

if $e \in AppExpr$ **then**
  for $i = 1$ *to* $e.args\_num$ **do**
    $BU(\Gamma,\ e.arg_i)$
  end for
  $\sum(e.constructor) = \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau_r$
  $\Gamma = \Gamma \cup \{e : F\ \tau_{e.arg_1}\ \tau_{e.arg_2}\ \cdots\ \tau_{e.arg_{args\_num}}\}$
  $Unify(\Gamma,\ e)$
else if $e \in ListExpr$ **then**
  for $i = 1$ *to* $e.exprs\_num$ **do**
    $BU(\Gamma,\ e.expr_i)$
  end for
  $\Gamma = \Gamma \cup \{e : \tau_{e.expr_1}\ list\}$
else if $e \in SetExpr$ **then**
  for $i = 1$ *to* $e.exprs\_num$ **do**
    $BU(\Gamma,\ e.expr_i)$
  end for
  $\Gamma = \Gamma \cup \{e : \tau_{e.expr_1}\ set\}$
else if $e \in LetInExpr$ **then**
  $BU(\Gamma,\ e.equation.expr)$
  $\Gamma = \Gamma \cup \{e.equation.pattern : \tau_{e.equation.expr}\}$
  $EX(\Gamma,\ e.equation.pattern)$
  $BU(\Gamma,\ e.expr)$
  $\Gamma = \Gamma \cup \{e : \tau_{e.expr}\}$
else if $e \in CaseExpr$ **then**
  $BU(\Gamma,\ e.expr)$
  for $i = 1$ *to* $e.equations\_num$ **do**
    $\Gamma = \Gamma \cup \{e.equation_i.pattern : \tau_{e.expr}\}$
    $EX(\Gamma,\ e.equation_i.pattern)$
    $BU(\Gamma,\ e.equation_i.expr)$
  end for
  $\Gamma = \Gamma \cup \{e : \tau_{e.equation_1.expr}\}$
else if $e \in LambdaExpr$ **then**

Algorithm 3 *Cont.*

for $i = 1$ *to* $e.parameters\_num$ **do**
  $\Gamma = \Gamma \cup \{e.patameter_i : \tau_{lambda@lambda\_counter}\}$
end for
$BU(\Gamma, e.expr)$
$\Gamma = \Gamma \cup \{e : \tau_{e.parameter_1} \Rightarrow \cdots \Rightarrow \tau_{e.parameter_{e.parameters\_num}} \Rightarrow \tau_{e.expr}\}$
$Unify(\Gamma, e)$
else if $e \in VarExpr$ **then**
  return
else if $e \in ConstExpr$ **then**
  if $e \in IntegralExpr$ **then**
    $\Gamma = \Gamma \cup \{e : nat\}$
  else
    $\Gamma = \Gamma \cup \{e : bool\}$
  end if
else
  $\Gamma = \Gamma \cup \{e : \bot\}$
end if

In the algorithm **BU**, type inference proceeds recursively. If an expression consists of several sub-expressions, **BU** is applied to infer their types, which are then stored in Γ. The relevant bottom-up rules are subsequently applied to infer the type of the entire expression. Because type inconsistencies may arise in *AppExpr* and *LambdaExpr*, the unification algorithm *Unify* is invoked to resolve them. If an expression is *Const* or *VarExpr*, its type has already been obtained by the pattern parameter type extraction module and stored in Γ, so no further action is required. If an expression does not belong to any of these categories, it is assigned the undefined type. Once all expressions on the right-hand side of the function equations have been processed, all type information is stored in the enriched AST.

5.3. Top-Down Type Completion Module

After the bottom-up type inference module, most type-missing problems can be resolved; however, issues remain in confirming the types of certain expressions. Assume that the type of the overall expression on the right-hand side of a function equation is τ_X. This type may be inconsistent with the return type τ_N specified in the function signature. This inconsistency arises because some terms in the expression retain their originally defined types, and the expression construction process cannot provide additional type information for their instantiation. As a result, the type of the final expression may remain partially uninstantiated. Consider the following specification *product_lists* as an example:

```
primrec product_lists :: "α list list => α list list" where
  "product_lists [] = [[]]" |
  "product_lists (xs # xss) = concat (map (λx.map (Cons x)
    (product_lists xss)) xs)"
```

In Isabelle/HOL, [] (a symbolic representation of *Nil*) has an originally defined type of *α list*. Suppose that the type of [] on the right-hand side of the first equation is modified to *α*#1 *list*. Then, according to the bottom-up type inference, the type of [[]] on the right-hand side becomes *α*#1 *list list*. However, according to the specification of *product_lists*, the return type of the expression [[]] should be *α list list*, resulting in a conflict.

In this case, type unification cannot resolve the inconsistency. This is because type unification proceeds from sub-expressions or parameters to the entire expression. In the present example, however, the inconsistency arises at the level of the entire expression, while the types of all sub-expressions or parameters remain locally consistent. Therefore,

after completing bottom-up type inference, it is necessary to use the type information provided by the function specification to perform top-down type completion on the right-hand side expression. The top-down type completion Algorithm 4 is presented below.

Algorithm 4 $TD(\Gamma,\ e)$

if $e \in AppExpr$ **then**
  $\sum(e.constructor) = \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau_r$
  $\Gamma \vdash e : \tau_e$
  if $\tau_r \succeq \tau_e$ *and* **then**
    $\tau_r \succeq \tau_e \overset{*}{\Longrightarrow} \mathcal{S}$
  else
    $\mathcal{S} = [\,]$
  end if
  for $i = 1$ *to* $e.args\_num$ **do**
    if $\tau_{e.arg_i} \succeq \mathcal{S}\tau_i$ **then**
      $\tau_{e.arg_i} = \mathcal{S}\tau_i$
    end if
    $TD(\Gamma,\ e.arg_i)$
  end for
else if $e \in ListExpr$ **then**
  $\Gamma \vdash e : \tau\ list$
  for $i = 1$ *to* $e.exprs\_num$ **do**
    $\Gamma = \Gamma \cup \{e.expr_i : \tau\}$
    $TD(\Gamma,\ e.expr_i)$
  end for
else if $e \in SetExpr$ **then**
  $\Gamma \vdash e : \tau\ set$
  for $i = 1$ *to* $e.exprs\_num$ **do**
    $\Gamma = \Gamma \cup \{e.expr_i : \tau\}$
    $TD(\Gamma,\ e.expr_i)$
  end for
else if $e \in LetInExpr$ **then**
  $\Gamma \vdash e : \tau$
  $\Gamma = \Gamma \cup \{e.expr : \tau\}$
  $TD(\Gamma,\ e.expr)$
else if $e \in CaseExpr$ **then**
  $\Gamma \vdash e : \tau$
  for $i = 1$ *to* $e.equations\_num$ **do**
    $\Gamma = \Gamma \cup \{e.equation_i.expr : \tau\}$
    $TD(\Gamma,\ e.equation_i.expr)$
  end for
else if $e \in LambdaExpr$ **then**
  return
else if $e \in VarExpr$ **then**
  return
else if $e \in ConstExpr$ **then**
  return
else
  $\Gamma = \Gamma \cup \{e : \bot\}$
end if

In the Algorithm 4, the type of the entire right-hand side expression in each equation is assigned to the return type specified in the function signature, and type inference is then propagated to its sub-expressions. For each expression, the algorithm determines the abstract–concrete relation between the type obtained via top-down type completion and that obtained via bottom-up inference. If the former is more abstract than the latter, type replacement is performed, and the sub-expressions are processed accordingly. After completing top-down type completion, the algorithm updates the type information in the enriched AST.

It should be noted that top-down type completion must be performed after bottom-up type inference. This is because the types of some expressions cannot be inferred from the types of their sub-expressions. For example, the expression $expr_1 = expr_2$ has type $bool$, but the types of its left and right sub-expressions cannot be derived from bool. There is no constraint relationship between the types of its sub-expressions and the type of the overall expression; such operators are typically polymorphic and defined by type classes (e.g., = and >). Therefore, if $AppExpr$ is a polymorphic operator such as '=', the complete type information cannot be inferred through top-down type completion alone, and the current result must be returned for further processing.

6. Experiment and Analysis

6.1. Case Study

The type system proposed in this paper addresses the problems of type unification and type inference for lambda expressions in the previous Isabelle2Cpp framework. With this type system, the Isabelle2Cpp generation framework can automatically obtain more complete type information and improve the C++ code generation process. This section illustrates the effectiveness of Isabelle2Cpp type inference by comparing the C++ code generated from the specifications of binary search *bs* and *product_lists* using the proposed type system.

In Isabelle/HOL, the specification *bs* for binary search is defined as follows:

```
fun bs :: "nat => nat list => nat option" where
  "bs x [] = None" |
  "bs x [y] = If (x = y) (Some 0) None" |
  "bs x ys = (let m = (length ys) div 2 in
      let y = ys ! m in
        If (y = x)
          (Some m)
          (If (y < x)
            (case bs x (drop (m + 1) ys) of Some n => Some
            (m + n + 1) |
                None => None)
            (bs x (take m ys)
          )
      )
  )"
```

After applying the proposed type inference algorithms, the C++ code generated by Isabelle2Cpp from *bs* is shown below. Statements generated before applying the proposed type inference system are marked with '−' at the beginning of the line, whereas those generated after applying the proposed system are marked with '+':

```
#include "binary_search.hpp"

std::optional<std::uint64_t> bs(const std::uint64_t &arg1,
                                std::deque<std::uint64_t> arg2) {
    // bs x [] = None
    if (arg2.empty()) {
        return std::optional<std::uint64_t>();
    }

    // bs x [y] = If (x = y) (Some 0) None
    if (arg2.size() == 1) {
        auto y = arg2[0];
        std::optional<std::uint64_t> temp0;
        if (arg1 == y) {
            temp0 = std::make_optional<std::uint64_t>(0);
        } else {
            temp0 = std::optional<std::uint64_t>();
        }
        return temp0;
    }

    // bs x ys = (let m = (length ys) div 2 in ...
    auto temp0 = arg2.size() / 2;
    auto m = temp0;
    auto temp2 = arg2;
    auto temp1 = temp2[m];
    auto y = temp1;
    std::optional<std::uint64_t> temp3;
    if (y == arg1) {
        temp3 = std::make_optional<std::uint64_t>(m);
    } else {
        std::optional<std::uint64_t> temp4;
        if (y < arg1) {
            auto temp5 = ([&] {
-               auto temp6 = bs(arg1, decltype(arg2)(arg2.begin() +
                            m + 1, arg2.end()));
+               auto temp6 = bs(arg1, std::deque<std::uint64_t>(arg2.begin() +
                            m + 1, arg2.end()));
                    ...
            })();
            temp4 = temp5;
        } else {
-           temp4 = bs(arg1, decltype(arg2)(arg2.begin(),
                    arg2.begin() + m));
+           temp4 = bs(arg1, std::deque<std::uint64_t>(arg2.begin(),
                    arg2.begin() + m));
        }
        temp3 = temp4;
    }
    return temp3;
}
```

For the generated code without the type system, the type of the second argument in the function calls corresponding to the temporary variables *temp*6 and *temp*4 is deduced in

C++ using **decltype** based on the type of *arg*2. With the proposed type system, however, Isabelle2Cpp can directly determine std::deque<std::uint64_t> as the parameter type. This demonstrates that the proposed type system reduces the dependence of the Isabelle2Cpp code generation framework on **auto** and **decltype**, thereby generating code with richer and more explicit type information.

Explicit type declarations offer several benefits:

1. They improve code readability. While **decltype**$(arg2)$ correctly deduces the type of *arg*2 (i.e., $std :: deque < std :: uint64_t >$), it requires the reader to infer the type of *arg*2. Writing $std :: deque < std :: uint64_t >$ explicitly makes the intent immediately clear and eliminates repeated contextual verification.
2. They reduce the risk of type deduction failures. In scenarios involving templates or nested namespaces, the operation of **decltype**$(arg2)$ relies on the declaration of *arg*2 within its scope. If *arg*2 is the result of a complex expression, the deduction rules of **decltype**, particularly when combined with **auto** and reference collapsing, may yield unexpected results. Explicit types help avoid errors that may arise from **auto**-based type inference.
3. They improve compilation speed. Although the overhead is typically small, parsing **decltype** expressions requires the compiler to perform type deduction; when used extensively, this can increase compilation overhead. Explicit types provide type information directly, which can improve compilation efficiency, especially in large-scale projects. This aligns with the C++ Core Guidelines, which recommends preferring explicit types over excessive reliance on **auto** and **decltype**.

In another example of *product_lists*, the types of [[]] and $\lambda\ x.\ map\ (Cons\ x)\ (product\_lists\ xss)$ cannot be inferred by C++ type inference. In addition, code generation for this function requires the type of the curried expression *Cons x* to be determined. With the proposed type system, it can be automatically inferred that the type of [[]] is $\alpha\ list\ list$, which is consistent with the return type of *product_lists*. Moreover, the type of $(Cons\ x)$ is inferred as the function type $\alpha\ list \Rightarrow \alpha\ list$, which results in the lambda expression being assigned the type $\alpha \Rightarrow \alpha\ list\ list$. The specific type inference results are shown below. In these results, each expression in the definition of *product_lists* is annotated at an appropriate level of granularity. Parentheses are used to group each expression with its type, and :: is used to separate the expression from its corresponding type.

```
product_lists
([(Nil :: α list)] :: (α list) list)
((concat ((map (λx.((map ((Cons (x :: α))) :: (α list => α list))
((product_lists (xss :: (α list) list)) :: (α list) list))
:: (α list) list) :: (α => (α list) list))(xs :: α list)) ::
((α list) list) list)) :: (α list) list)
```

These cases demonstrate that the proposed type system eliminates Isabelle2Cpp's dependence on C++ type inference. It provides type inference for pattern parameters and a wide range of expressions, thereby offering effective support for Isabelle2Cpp code generation.

6.2. List.thy

The library file List.thy contains 78 Isabelle/HOL specifications for datatypes and functions, including 1 datatype definition and 77 function definitions. Of these 78 functions, 50 successfully pass type inference. Among the remaining functions, 18 involve set operations and cannot be processed by Isabelle2Cpp for code generation; these are therefore excluded from type inference. The remaining 10 functions use advanced functional features,

namely higher-order functions, which are currently unsupported by both the type inference system and the code generator. The experimental results are shown in Table 1.

Table 1. Conversion results for List.thy after incorporating the type system and reasons for failures.

Function Definitions in List.thy	Type Inference	Illustration
(50) *'a list*, *last*, *butlast*, *append*, *rev*, *filter*, *foldl*, *concat*, *drop*, *take*, *nth*, *list_update*, *takeWhile*, *dropWhile*, *zip*, *upt*, *insert*, *find*, *count_list*, *removel*, *removeAll*, *distinct*, *remdups*, $remdups_adj$, *replicate*, *length*, *enumerate*, *rotatel*, *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*, *n_lists*, *map*2, *subseqs*, *those*, *product*, $product_lists$	Succeeded	
(18) *coset*, *sorted_wrt*, *sorted*, *strict_sorted*, *stable_sort_key*, *lex*, *lenlex*, *lexord*, *listrell*, *list_exl*, *can_select*, *listrellp*, *lexordp*, *set_Cons*, *lexn*, *all_interval_nat*, *all_interval_int*, *map_project*	Failed	The set operations involving the selection axiom cannot be used for code generation by Isabelle2Cpp.
(10) *rotate*, *fold*, *foldr*, *sort_key*, *union*, *nths*, *insort_insert_key*, *map_filter*, *extract*, *sorted_list_of_set*	Failed	Isabelle2Cpp currently does not support advanced functional features.

6.3. Scalability and Complexity Analysis

The code generator provided by Isabelle/HOL can directly generate code for a variety of functional programming languages, including SML, OCaml, Haskell, and Scala. Unlike these generators, Isabelle2Cpp converts functional specifications into imperative C++. The type inference rules and unification algorithm proposed in this paper are specifically designed for the Isabelle2Cpp code generation framework. The proposed system avoids both the constraint solving typical of standard HM type systems and the type coercions present in Isabelle/HOL, thereby reducing the complexity of type inference.

The input to the proposed type inference system must consist of formal specifications verified in Isabelle, whose correctness is guaranteed by Isabelle/HOL. Consequently, inference failures caused by invalid or ambiguous inputs do not occur. Specifications that cannot be executed in Isabelle/jEdit are not acceptable as input to the system. The limitations of the proposed method primarily stem from the parsing and code generation constraints of the Isabelle2Cpp framework. Certain set-related operations and higher-order functional features are not supported, which limits the applicability of this type system.

The runtime of the type inference algorithm scales linearly with the length of the expression. The process involves recursive operations and incurs memory overhead, which has some impact on the efficiency of the code generation phase. However, this cost is necessary: introducing the type system does not degrade the execution efficiency of the generated C++ code; instead, it contributes to producing more complete and accurate code. By applying the proposed type inference system and algorithm, Isabelle2Cpp can obtain more comprehensive type information and, consequently, generate more complete C++ code.

7. Conclusions

This paper proposes a type system for the Isabelle2Cpp code generation framework that is designed to perform type inference and type unification on expressions in the intermediate representation. The system comprises three cooperating modules: the pattern parameter type extraction module, the bottom-up type inference module, and the top-

down type completion module. The pattern parameter type extraction module obtains type information for variables other than functions and datatype constructors, thereby providing a reliable source of type information. The bottom-up type inference module derives detailed type information for all expressions and performs type unification. The top-down type completion module integrates the type information obtained from bottom-up type inference with that provided by the function specification and performs type consistency checking for the entire expression.

Compared with existing works, the type system proposed in this paper is based on the HM type system and is specifically designed for the Isabelle2Cpp code generation framework. Inference rules and unification algorithms are introduced to extend the Isabelle2Cpp framework by incorporating a type system and enabling complete type inference. The design minimizes modifications to the standard HM type system and avoids introducing additional mechanisms, such as constraint solving or coercion, thereby reducing system complexity. By incorporating this type system, Isabelle2Cpp provides more comprehensive type information for expressions and can automatically generate more complete and accurate C++ code, even when some type information is missing from the Isabelle/HOL specification.

Author Contributions: Conceptualization, D.J.; methodology, D.J. and C.F.; software, C.F. and Y.M.; formal analysis, D.J. and Y.M.; investigation, C.F. and Y.M.; writing—original draft preparation, D.J. and C.F.; writing—review and editing, D.J. and Y.M.; supervision, D.J.; funding acquisition, Y.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Isabelle2Cpp can be downloaded from https://github.com/jasonme0830/Isabelle2Cpp-2025.git (accessed on 11 April 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Nipkow, T.; Wenzel, M.; Paulson, L.C. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*; Springer: Berlin/Heidelberg, Germany, 2002.
2. Jiang, D.; Xu, B, Generation of C++ Code from Isabelle/HOL Specification. *Int. J. Softw. Eng. Knowl. Eng.* **2022**, *32*, 1043–1069. [CrossRef]
3. Cardelli, L. Type systems. *ACM Comput. Surv.* **1996**, *28*, 263–264. [CrossRef]
4. Duggan, D.; Bent, F. Explaining type inference. *Sci. Comput. Program.* **1996**, *27*, 37–83. [CrossRef]
5. Girard, J. The system F of variable types, fifteen years later. *Theor. Comput. Sci.* **1986**, *45*, 159–192. [CrossRef]
6. Milner, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.* **1978**, *17*, 348–375. [CrossRef]
7. Damas, L. *Type Assignment in Programming Languages*; KB Thesis Scanning Project; The University of Edinburgh, Old College: Edinburgh, UK, 2015; p. 1984.
8. Soosaipillai, H. An Explanation Based Polymorphic Type Checker for Standard ML. Master's Thesis, Heriot-Watt University, Currie, UK, 1990.
9. Heeren, B.J.; Hage, J.; Swierstra, S.D. *Generalizing Hindley-Milner Type Inference Algorithms*; Information and Computing Sciences, Utrecht University: Utrecht, The Netherlands, 2002.
10. Kfoury, A.J.; Wells, J.B. Principality and type inference for intersection types using expansion variables. *Theor. Comput. Sci.* **2004**, *311*, 1–70. [CrossRef]
11. Vytiniotis, D.; Jones, S.P.; Schrijvers, T.; Sulzmann, M. OutsideIn (X) Modular type inference with local assumptions. *J. Funct. Program.* **2011**, *21*, 333–412. [CrossRef]
12. Traytel, D.; Berghofer, S.; Nipkow, T. Extending Hindley-Milner type inference with coercive structural subtyping. In *Programming Languages and Systems: 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, 5–7 December 2011, Proceedings 9*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 89–104.

13. Palsberg, J.; Schwartzbach, M.I. Object-oriented type inference. *ACM Sigplan Not.* **1991**, *26*, 146–161. [CrossRef]
14. Pierce, B.C.; Turner, D.N. Local type inference. *ACM Trans. Program. Lang. Syst. (Toplas)* **2000**, *22*, 1–44. [CrossRef]
15. Chen, L.; He, Z.; Mao, B. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*; IEEE: Piscataway, NJ, USA, 2020; pp. 88–98.
16. Hellendoorn, V.J.; Bird, C.; Barr, E.T.; Allamanis, M. Deep learning type inference. In *Proceedings of the 2018 26th Acm Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*; Association for Computing Machinery (ACM): New York, NY, USA, 2020; pp. 152–162.
17. Krauss, A. Defining Recursive Functions in Isabelle/HOL. 2008. Available online: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://isabelle.in.tum.de/website-Isabelle2021-1/dist/Isabelle2021-1/doc/functions.pdf&ved=2ahUKEwiEgNyKg_yTAxWskq8BHR7xJBwQFnoECBwQAQ&usg=AOvVaw1_oYLdE5ure9Z4l3FSzA9H (accessed on 11 April 2026).