Electronic Communications of the EASST
Volume 10 (2008)

Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)

Type Checking C++ Template Instantiation by Graph Programs

Karl Azab and Karl-Heinz Pennemann

14 pages

Guest Editors: Claudia Ermel, Reiko Heckel, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Type Checking C++ Template Instantiation by Graph Programs

Karl Azab and Karl-Heinz Pennemann

azab@informatik.uni-oldenburg.de, pennemann@informatik.uni-oldenburg.de
Carl v. Ossietzky Universität Oldenburg, Germany

Abstract: Templates are a language feature of C++ and can be used for metaprogramming. The metaprogram is executed by the compiler and outputs source code which is then compiled. Templates are widely used in software libraries but few tools exist for programmers developing template code. In particular, error messages are often cryptic. During template instantiation, a compiler looks up names that depend on a template's formal parameters. We use graphs to represent the relevant parts of the source code and a graph program for the name lookup and type checking for expressions involving such names. This technique provides compiler writers with a visual way of writing algorithms that generate error messages and forms the basis for a visual inspection of type problems and suggested remedies for the programmer. Our graph program terminates and emits correct error messages.

Keywords: Graph programs, Type checking, C++

1 Introduction

Templates are a feature of the C++ programming language for generic programming, i.e. programmed code generation. Generic source code is written by omitting the specific data types of variables and instead supplying those as parameters (*parameterized types*). A parameterized type and variable of that type can be used as any other type or variable, e.g. the type name can be used to resolve names and the variable's members can be accessed. This way, templates separate types from algorithms in design, and combines them into new class-types and functions at compile time. Compared to non-template code which uses a generic type like `void *`, an immediate advantage from templates is improved static type checking. Templates are used extensively in the Standard Template Library and Boost libraries [Jos99, AG04]. They have also found use in performance critical domains, such as scientific computing and embedded systems [Vel98, Str04]. An introduction to templates can be found in e.g. [Str00].

A class type or function containing generic source code is called a *template definition*. A list of type parameters for a particular template definition is called a *declaration*. For each unique declaration, the *template instantiation* mechanism generates a specialization of that template definition. A *specialization* is a copy of the definition where the parameterized types are replaced by the declaration's actual type parameters. Non-types, i.e. constants, are allowed as template parameters, allowing e.g. array sizes to be set at compile time. Templates form a computationally complete *metalanguage* [CE00], a sub-language of C++ executed during compilation.

Consider the following example: A parameterized type is used to resolve the name `size` in the template definition in Figure 1. The first specialization is for the declaration `icon<char>` and will not compile since the provided type `char` has no field named `size` and can therefore

not be used for the expression defining the array size. For the second specialization, if the type `resolution<128>` contains a static field named `size` of an unsigned integer type, then the second specialization will compile.

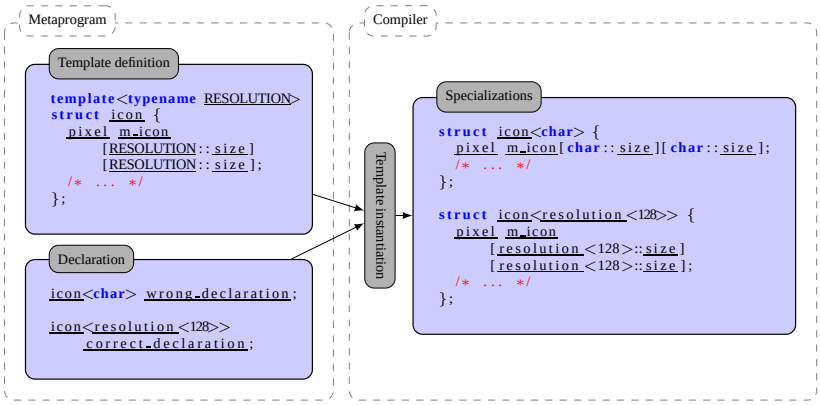


Figure 1: C++ template instantiation.

Even though templates are a useful technique, they can be complex and difficult to read and write. In particular, error messages are often cryptic. This has led to the development of methods and tools to analyze template code. The usage of specializations can be analyzed by debuggers, software patterns like tracers [VJ02], and tools like TUAnalyzer [GPG04]. For the metaprogram itself, research is being done on a debugging framework Templight [PMS06].

To improve error messages, we suggest modeling definitions and declarations by graphs, while name lookup and type checking of such graphs is made by graph programs that emit error messages as graphs instead of text. Graphs allow an abstract and visual representation of all necessary information, while graph programs provide an intuitive way of writing programs that detect problems and suggests remedies. In combination with presentation techniques such as focusing (on relevant parts) and hierarchical depiction, we believe that our model is usable as a basis for a visual inspection of type problems and suggested remedies.

Graph transformation systems is a well investigated area in theoretical computer science. An overview on the theory and applications is given in the book *Fundamentals of Algebraic Graph Transformation* [EEPT06]. Graph transformation systems rewrite graphs with (graph transformation) rules. A rule describes a left- and right-hand side. A transformation step is done by matching the left-hand side to a subgraph of the considered graph and modifying that subgraph according to the difference of the left- and right-hand side. Graph programs [HP01, PS04] provide a computationally complete programming language based on graph transformations. Graph conditions [HP05] can be used to express properties of graphs by demanding or forbidding the existence of specific structures. In a similar way, graph conditions can limit the applicability of rules in a graph program, by making demands on elements local to the subgraph matched by a rule.

In this paper, we use graphs to represent the template source code necessary for name lookup and type checking during template instantiation. We refer to those graphs as source-code graphs.

A graph program TTC (*Template-Type Checker*) looks up dependent names and detects type clashes in expressions for a subset of the C++ template features. TTC attempts to solve type clashes by implicit type casts. If such a cast loses precision, a warning message is generated. If no appropriate cast is found, an error message is generated, indicating the location of the error and suggesting a remedy for the programmer. TTC outputs a message graph, where errors and warnings are embedded. The message graph is interpreted by the programmer with the help of graph conditions. Graph conditions detect warning and error messages in graphs and when an error is present, they can determine for which declarations a definition can successfully be instantiated. Figure 2 gives an overview.

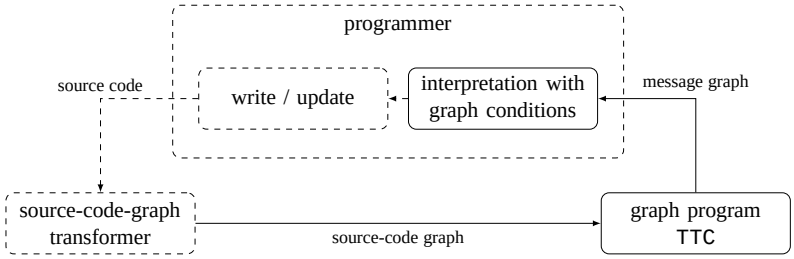


Figure 2: TTC type checks graphs and outputs error messages.

The paper is structured as follows. Graph programs are introduced in Section 2. Section 3 informally describes how C++ source code is transformed into source-code graphs and defines type safety for graphs. In Section 4 we present the graph program TTC for transforming a source-code graph into a message graph. In Section 5 we give proof ideas for how to show that TTC terminates and that the error messages generated by it correctly indicate that the input is not type safe. We conclude our results in Section 6. A long version of this paper, with complete proofs and more examples, is available as a technical report, see [AP07].

2 Graph Programs

In this section, we review conditions, rules, and programs, in the sense of [HP05] and [HP01]. In the following, we consider the category of directed labeled graphs with all injective graph morphisms. Labels distinguish different types of nodes and edges and directions model relationships between nodes. We use the standard definition of labeled graphs and labeled graph morphisms, see [EEPT06] or [AP07] for details. For expressing properties on graphs we use so-called graph conditions. The definition is based on graph morphisms.

Definition 1 (Graph conditions) A *graph condition* over an object P is of the form $\exists a$ or $\exists(a,c)$, where $a\colon P \to C$ is a morphism and c is a condition over C. Moreover, Boolean formulas over conditions (over P) are conditions (over P). A morphism $p\colon P \to G$ *satisfies* a condition $\exists a$ ($\exists(a,c)$) over P if there exists an injective morphism $q\colon C \to G$ with $q \circ a = p$ (satisfying c). An object G *satisfies* a condition $\exists a$ ($\exists(a,c)$) if all injective morphisms $p\colon P \to G$ satisfy the

condition. The satisfaction of conditions over P by objects or morphisms with domain P is extended to Boolean formulas over conditions in the usual way. We write $p \models c$ ($G \models c$) to denote that morphism p (object G) satisfies c. In the context of rules, conditions are called *application conditions*.

We rewrite graphs with rules in the double-pushout approach [EEPT06]. Application conditions specify the applicability of a rule by restricting the matching morphism.

Definition 2 (Rules) A *plain rule* $p = \langle L \leftarrow K \rightarrow R \rangle$ consists of two injective morphisms with a common domain K. L is the rule's left-hand side, and R its right-hand side. A *left application condition* ac for p is a condition over L. A *rule* $\hat{p} = \langle p, \mathrm{ac} \rangle$ consists of a plain rule p and an application condition ac for p.

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow{\scriptstyle m^*} \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Given a plain rule p and injective morphism $K \rightarrow D$, a *direct derivation* consists of two pushouts (1) and (2) where the *match* m and *comatch* m^* are required to be injective. We write a direct derivation $G \Rightarrow_{p,m,m^*} H$. Given a graph G together with an injective match $m : L \rightarrow G$, the direct derivation $G \Rightarrow_{p,m,m^*} H$ can informally be described as: H is obtained by deleting the image $m(L-K)$ from G and adding $R-K$. Given a rule $\hat{p} = \langle p, \mathrm{ac} \rangle$ and a morphism $K \rightarrow D$, there is a *direct derivation* $G \Rightarrow_{\hat{p},m,m^*} H$, if $G \Rightarrow_{p,m,m^*} H$, and $m \models \mathrm{ac}$.

We now define graph programs as introduced in [HP01].

Definition 3 (Graph programs) Every rule p is a *(graph) program*. Every finite set $\mathscr{S}$ of programs is a program. If P and Q are programs, then $(P;Q)$, P^* and $P{\downarrow}$ are programs. The *semantics* of a program P is a binary relation $[\![P]\!] \subseteq \mathscr{G}_{\mathscr{C}} \times \mathscr{G}_{\mathscr{C}}$ on graphs: (1) For every rule p, $[\![p]\!] = \{\langle G,H \rangle \mid G \Rightarrow_p H\}$. (2) For a finite set $\mathscr{S}$ of programs, $[\![\mathscr{S}]\!] = \cup_{P \in \mathscr{S}} [\![P]\!]$. (3) For programs P and Q, $[\![(P;Q)]\!] = [\![Q]\!] \circ [\![P]\!]$, $[\![P^*]\!] = [\![P]\!]^*$ and $[\![P{\downarrow}]\!] = \{\langle G,H \rangle \in [\![P]\!]^* \mid \neg \exists M. \langle H,M \rangle \in [\![P]\!]\}$.

Programs according to (1) are *elementary* and a program according to (2) describes the *nondeterministic choice* of a program. The program $(P;Q)$ is the *sequential composition* of P and Q. P^* is the *reflexive, transitive closure* of P, and $P{\downarrow}$ the *iteration* of P as long as possible. Programs of the form $(P;(Q;R))$ and $((P;Q);R)$ have the same semantics and are considered as equal; by convention, both can be written as $P;Q;R$. We use $P{\downarrow}^+$ as a shortening of $P;P{\downarrow}$.

Notation. When the label of an element is either a or b we use the notation $a|b$. $\langle L \Rightarrow R \rangle$ is used a short form of $\langle L \leftarrow K \rightarrow R \rangle$, where K consists of the elements common to L and R. For an application condition with morphism $a\colon P \rightarrow C$, we omit P as it can be inferred from the left-hand side. We omit the application condition if it is satisfied by any match. To distinguish nodes with the same label, we sometimes add an identifier in the form of "label:id". We use source-code fragments as identifiers and therefore print them in a fixed-width font.

3 From Source Code to Source-Code Graphs

In this section, we introduce source-code graphs, the input for our type-checking program, and informally describe how source code is transformed into such graphs. A source-code graph is a graph representation of template definitions, declarations, and expression's specializations. The type signature of every declared method, function, and operator in a template definition is represented in the graph by an overload forest, see below. Expressions that involve parameterized types are represented in the graph by expression paths, explained shortly. For every declaration, the above mentioned graph representations are copied and the parameterized types are replaced by the actual types provided by the declaration.

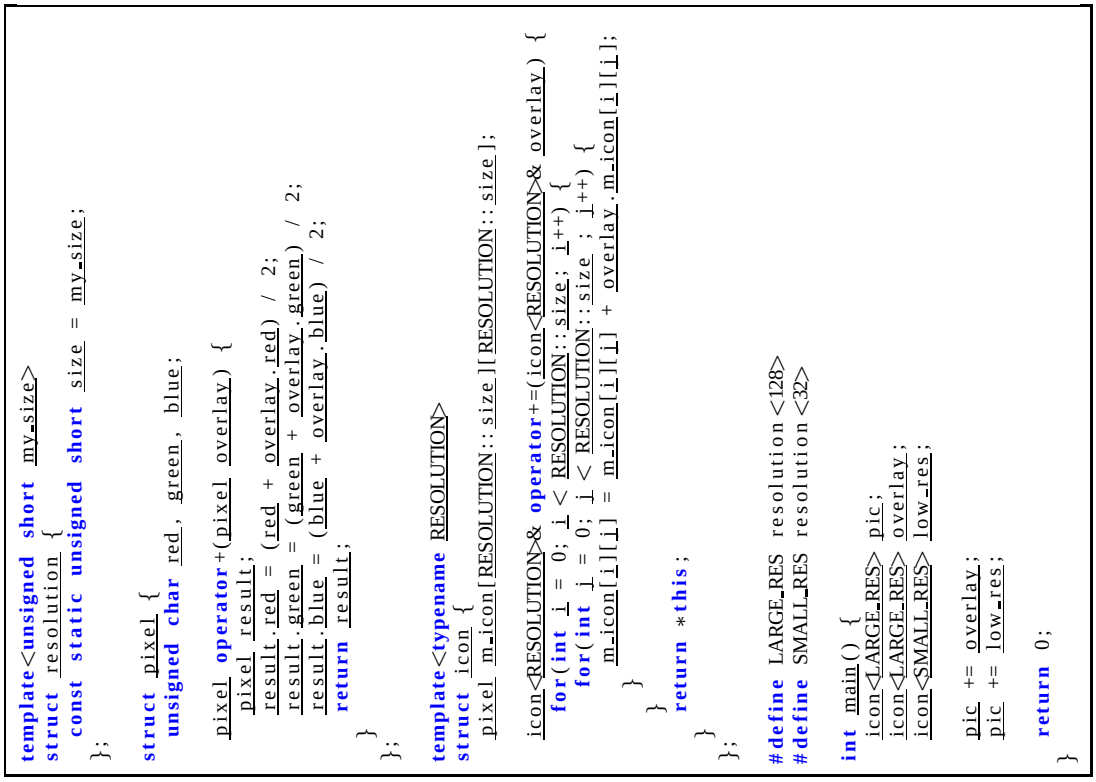
The basic elements of our source-code graphs are nodes for template definitions, declarations, data types, names, type signatures, and expression trees. For quick reference, node and edge labels together with a short explanation are listed below. Note that a visualization of an edge as dashed or solid denotes a difference in labels.

Nodes				Edges	
D	declaration	p	actual parameter	=	comparison
E	error message	t	data type	c	cast without precision loss
ex	(sub)expression	T	template definition	d	deduced type of expression
ol	overloaded operator	W	warning message	p	parameter
op	operator name			pc	cast with precision loss
				r	return type
				R	recovered comparison

Template definitions are represented by T-nodes. Two declarations are equivalent if they are based on the same template and have equal lists of template parameters. Each class of equivalent declarations are represented by a D-node and denotes a future specialization. Each D-node has an incoming edge from the T-node representing the template definition the declaration means to instantiate. Possible template parameters are represented by t-nodes. Such parameters include classes, structures, fundamental types, and constants. Operator-, function- and method names are represented by op-nodes. In [AP07] we show a graph program that generates source-code graphs.

Example 1 *Consider the source code with two class-type templates, line 1 and 18, in Figure 3. The principal data type is the* `icon` *structure with a template parameter for its resolution type. The* `resolution` *structure has a constant template parameter for a (quadratic) resolution. For the two unique declarations in* `main`*, name lookup and type checking is needed for the expressions on lines 3, 20, 23, 24, 25, 40, and 41. In Section 4 we will show how the graph program* TTC *reports the type clash in the expression on line 41. Note that Figure 4 shows the source-code graph of the source code example from Figure 3 where the above mentioned lines are represented as expression paths. That source-code graph also shows the overload trees for the operators on line 3 and 22. For completeness, some overload paths representing used operations native to C++ are also included, e.g. comparison (<) of integers.*

We will now introduce some necessary graph-theoretic notions. In particular, we introduce and use expression paths and overload trees to define type safety for graphs. Expression paths rep-



```
1  template<unsigned short my_size>
   struct resolution {
3    const static unsigned short size = my_size;
   };

5
   struct pixel {
7    unsigned char red, green, blue;

9    pixel operator+(pixel overlay) {
       pixel result;
11     result.red = (red + overlay.red) / 2;
       result.green = (green + overlay.green) / 2;
13     result.blue = (blue + overlay.blue) / 2;
       return result;
15   }
   };
17
   template<typename RESOLUTION>
19 struct icon {
     pixel m_icon[RESOLUTION::size][RESOLUTION::size];
21
     icon<RESOLUTION>& operator+=(icon<RESOLUTION>& overlay) {
23     for(int i = 0; i < RESOLUTION::size; i++) {
         for(int j = 0; j < RESOLUTION::size ; j++) {
25         m_icon[i][j] = m_icon[i][j] + overlay.m_icon[i][j];
         }
27     }
       return *this;
29   }
   };
31
   #define LARGE_RES resolution<128>
33 #define SMALL_RES resolution<32>

35 int main() {
     icon<LARGE_RES> pic;
37   icon<LARGE_RES> overlay;
     icon<SMALL_RES> low_res;
39
     pic += overlay;
41   pic += low_res;

43   return 0;
   }
```

Figure 3: Two class-type templates.

resent the type information from an expression tree and are modeled by ex- and p-nodes. The root of the tree becomes an ex-node and has an incoming edge from the D-node that represents the specialization in which it will exist. Each ex-node has an edge to the op-node denoting the operation's name. We allow for operators with an arbitrary number of operands, so the children of the root in an expression tree are modeled by a path of p-nodes. If such a child is a subexpression, then the corresponding p-node has an edge to a new expression node. If it is not, then it denotes a type and its p-node has an edge to the t|D-node denoting that type.

Definition 4 (Expression paths) Given a graph G and a natural number i, an *i-expression path* in G is a path $ex\,p_0 \dots p_i$, where the head, ex, is an ex-node and $p_0, \dots, p_i$ are p-nodes such that, from every node p_k, $0 \leq k < i$, the only edge to another p-node is to p_{k+1}.

Example 2 *Figure 5 shows (to the left) an expression tree denoting line 41 in Example 1 together with its corresponding* 2*-expression path (to the right).*

We represent the type signatures of methods with overload forests, trees and paths. A method named `method` declared in class type `class` with n parameters is represented by a path of $n+2$ ol-nodes. The head of that path has an edge to the op-node representing the name `method`

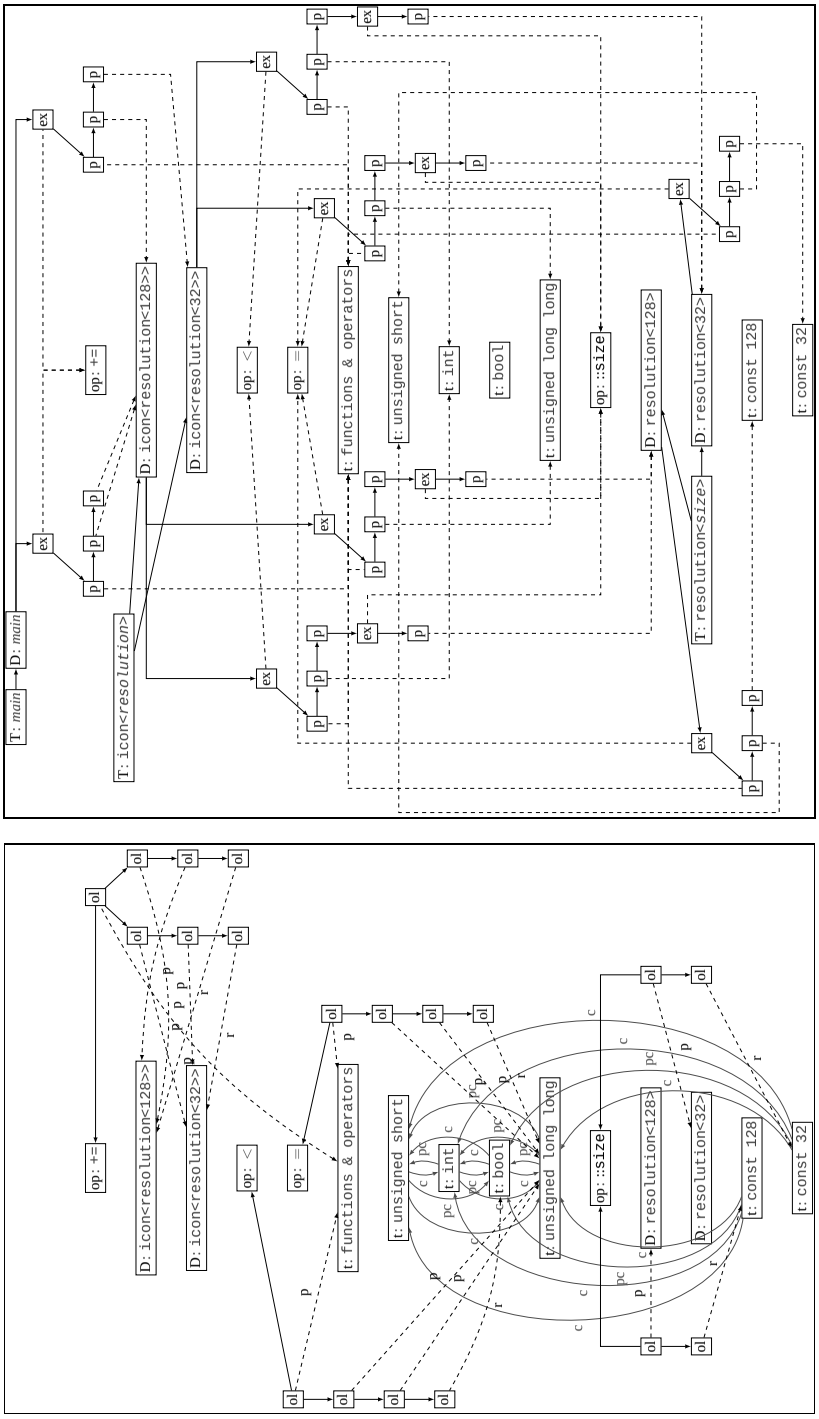


Figure 4: A source-code graph split in two for simpler representation, but note that the two subgraphs are not disjoint: the nodes with identical ids (see the center column) are identified.

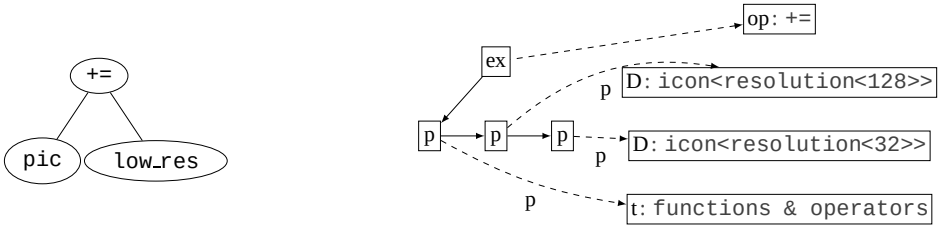


Figure 5: Expression paths represent expression trees.

and another edge to the t|D-node representing `class`. The ol-node at position k ($2 \leq k \leq n+1$) in the path has an edge to the node denoting the type of the variable at parameter position $k-1$. The last ol-node in the path has an edge to the t|D-node denoting the return type of `method`. Functions are modeled as methods but as declared in a special class with a name not allowed in the source language, e.g. `functions & operators`. Operator overloading is modeled as functions. In the following operators, methods, and functions are collectively referred to as *operators*.

Definition 5 (Overload forest) A graph G contains an *overload forest* iff all ol-nodes in G are part of exactly one overload tree and there exist no pair of overload trees with equivalent roots, see below. An *overload tree* is a maximal connected subgraph T in G, consisting of only ol-nodes. T is maximal in the sense that, if an ol-node in T has an edge to an ol-node, then that node is also in T. Furthermore, T must have a tree structure, i.e. no cycles and every node has one parent, except for the root. For nodes in T the following holds for them in G: (1) Each internal (leaf) node has exactly one p-edge (r-edge) to a t|D-node, one edge from its parent, and no other incoming edges. (2) The root of T has an additional edge to an op-node. (3) No two siblings have a p-edge to the same t|D-node. (4) Every node has at most one child that is a leaf. Requirements 3 and 4 are necessary to prevent ambiguous type signatures. Two roots are *equivalent* iff there exists an op-node o and t|D-node t, such that both roots have edges to o and t. An *i-overload path* $o_0 \dots o_{i+1}$ is a path in T from the root to a leaf. The t|D-node to which an r-edge exist from o_{i+1} is called the *return type* of the i-overload path.

Example 3 *The overload tree in Figure 6 has two* 2*-overload paths, representing the type signatures of two overloaded operators. The tree represents the operator template on line 22 and the two paths are generated for the two declarations on line 40 and 41 in Figure 3.*

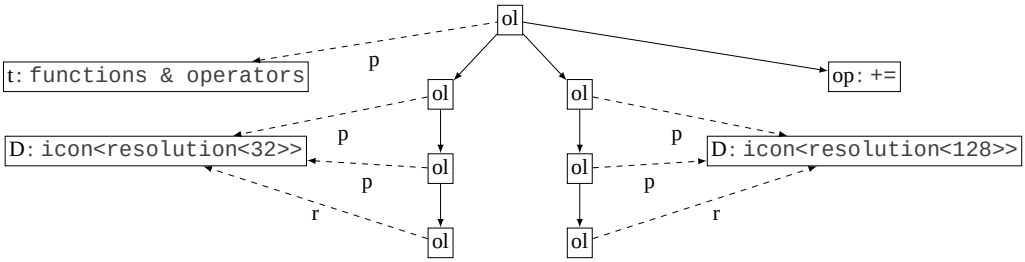


Figure 6: Two overload paths.

Remark 1. The size of a source-code graph grows linearly with the size of the source code that would be output by the template instantiation mechanism in a C++ compiler. An expression or declared operator that exists in such source code is represented only by a single expression path or overload path, respectively.

The main property in this paper is the one of type safety. A graph is type safe if for every expression path, there exists an overload path with the same type signature. This property corresponds to type safety for template instantiation in C++ programs, where every generated expression must be type checked against the existing and generated operators.

Definition 6 (Type-safe graphs) A graph G is *type safe* iff it contains an overload forest and is i-type safe for all natural numbers i. G is *i-type safe* iff every i-expression path in G is type safe. An i-overload path $o_0 \dots o_{i+1}$ makes the i-expression path $ex\, p_0 \dots p_i$ *type safe* iff:

1. There exists an op-node op and two edges: one from ex to op, the other from o_0 to op.
2. For all k, where $0 \leq k \leq i$, there exists a t|D-node t and two edges, one from o_k to t and the other is from p_k to either t or the head of a type safe j-expression path such that t is the deduced type of the that j-expression path.

The *deduced type* of the i-expression path is the t|D-node with an incoming r-edge from o_{i+1}.

It is easy to see that no overload path from Figure 6 makes the expression path in Figure 5 type safe.

4 The Type-Checking Program

This section describes the graph program `TTC` which performs the name lookup and type checks source-code graphs. The section also shows how message graphs are interpreted with graph conditions.

A schematic of how the subprograms of `TTC` interact is shown in Figure 7. Intuitively, `TTC` works as follows: The input is a source-code graph, each expression path is marked by `MarkExpression`, and `Compare` moves this marker through the path until it reaches the tail or a type clash. At a type clash, `Recover` either solves it or generates an error message. For markers at a tail, `Resolve` finds the deduced type of the expression path (i.e. it resolves the type of a subexpression). This chain is then iterated as long as new work is made available by `Recover` and `Resolve` for `Compare`. The yield of `TTC` is a message graph. Programs and rules are described in more detail below.

Definition 7 (`TTC`) Let the graph program `TTC` = `MarkExpression`↓;`TypeCheck`↓ with the subprograms:

$$
\begin{array}{lcl}
\texttt{TypeCheck} & = & \texttt{Compare}\downarrow^{+};\texttt{Recover}\downarrow;\texttt{AfterRecover}\downarrow;\texttt{Resolve}\downarrow \\
\texttt{Compare} & = & \{\ \texttt{Lookup}, \texttt{CompareNext}, \texttt{FindType}\ \} \\
\texttt{Recover} & = & \{\texttt{Cast}, \texttt{Warning}, \texttt{Error}\} \\
\texttt{Resolve} & = & \left\{ \begin{array}{c} \texttt{ResolveSubexpression}, \\ \texttt{ResolveExpression1}, \texttt{ResolveExpression2} \end{array} \right\}
\end{array}
$$

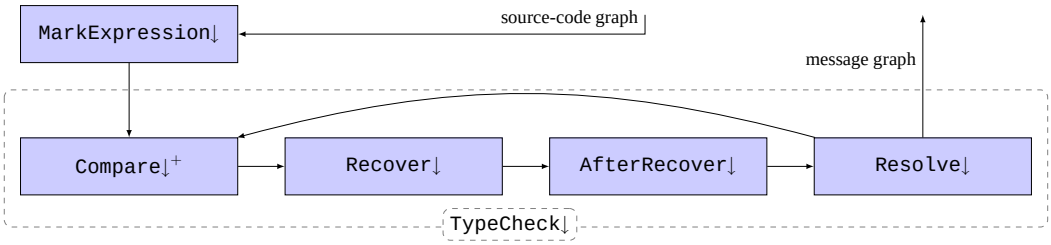
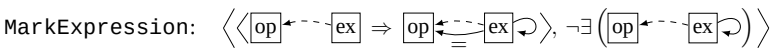


Figure 7: Structure of TTC.

`MarkExpression`↓ is executed before the actual type checking starts and markss each ex-node with an =-edge to show where the evaluation starts. To avoid duplicate evaluation, a loop is also placed at the ex-node and checked for in the application condition.



`Compare`↓+ consists of the rules `Lookup`, `CompareNext`, and `FindType`, see Figure 8. They move the =-edge generated by `MarkExpression` through the expression path as long as a matching overload path can be found. The program halts when it completes this matching or encounters a problem: that no overload path makes this expression path type safe or that this node in the path depends on the deduced type of a subexpression. The rule `Lookup` finds the overload tree for a marked expression's name. `CompareNext` matches the type signature of the expression path to an overload path parameter by parameter. The rule `FindType` is applied at the tail of the expression path and deduces the expression's type via the matched overload path's return type. The rule's application condition makes sure that this is actually the tail of the expression path.

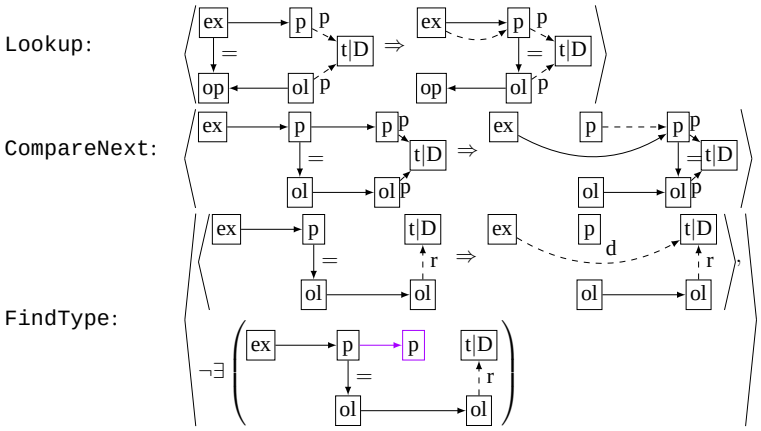


Figure 8: Rules in the program `Compare`.

`Recover`↓ consists of the rules Figure 9 and tries to find alternative overload paths by implicit type casts. The rule `Cast` finds a type cast that causes no loss of precision. `Warning` works as `Cast` but uses a cast with a possible loss of precision. For this we generate a warning, a W-node with three outgoing edges: the location of the problem, the original type, and the cast type. The application condition make sure that `Cast` is preferred. The rule `Error` is applied when there is no solution by implicit type casts. An error node is therefore generated. It has three outgoing edges, to the p-node where it occurred, the faulting type, and a suggested type. The application condition limits `Error` from being applied where `Cast` or `Warning` could be applied instead.

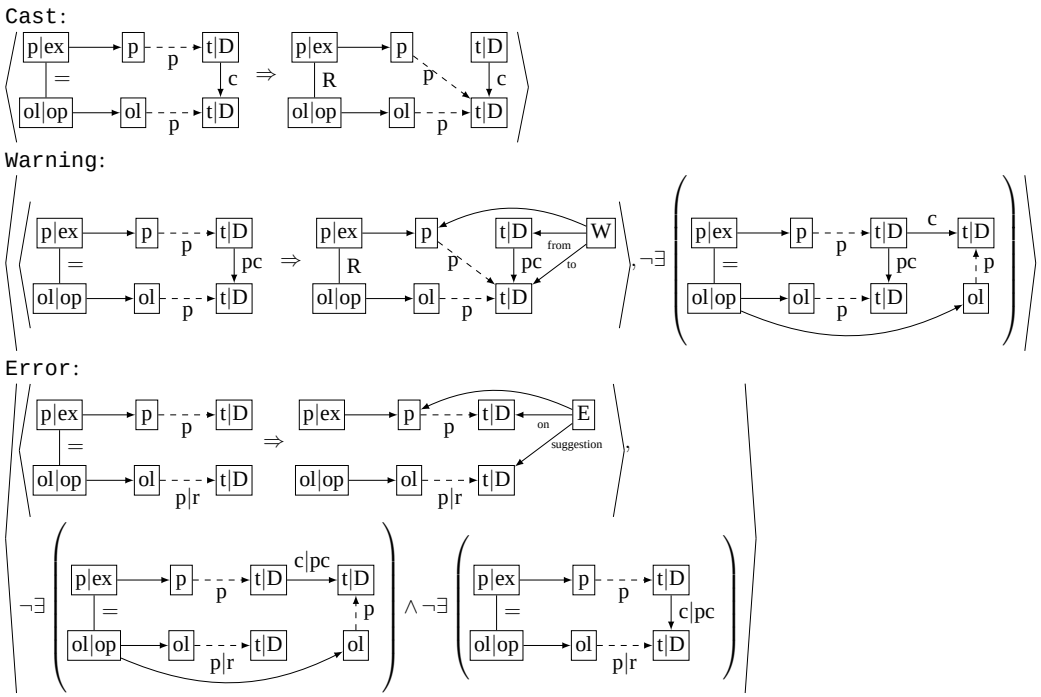
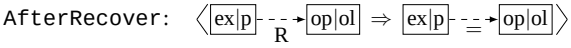


Figure 9: Rules for the program `Recover`.

`AfterRecover`↓ performs some cleanup work for `Recover`↓, generated R-edges are reset to =-edges.



`Resolve`↓ consists of three rules, as shown in Figure 10: `ResolveSubexpression` replaces a subexpression with its deduced return type. `ResolveExpression1` marks an expression as evaluated with a dashed edge. `ResolveExpression2` does the same in the special case when the return type is the same as the specialization in where the expression occurs.

Example 4 *The graph in Figure 11 is the yield of* `TTC` *applied to the overload tree from Figure 6 and the expression path from Figure 5. See [AP07] for more examples.*

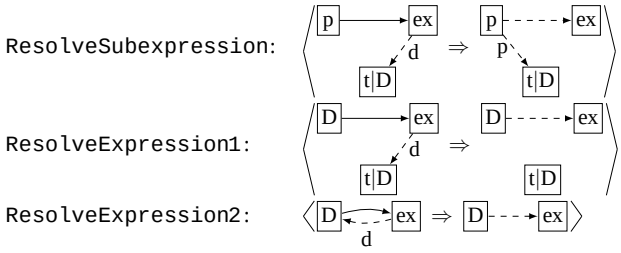


Figure 10: Rules in `Resolve`.

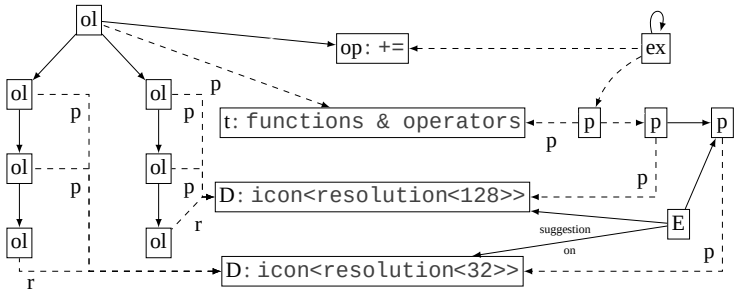


Figure 11: Portion of a message graph.

Remark 2. After the termination of `TTC`, graph conditions can help to interpret the message graph. A graph is an *error (warning)* graph iff it satisfies the condition $\exists(\emptyset \to \boxed{\text{E}})$ ($\exists(\emptyset \to \boxed{\text{W}})$). A particular declaration can safely be instantiated if its corresponding D-node satisfies the condition $\neg\exists(\boxed{\text{D}} \to \boxed{\text{D}} \longrightarrow \boxed{\text{ex}})$. If that condition is not satisfied, then one of its expressions could not be resolved and the programmer must take appropriate actions. A message graph will contain all the detected errors for the corresponding source code. Graph conditions can therefore help programmers to locate the areas of the graph that contain such errors. An implementation of this approach should be able to highlight the areas containing the errors.

Remark 3. The size of the resulting message graph will not grow more than linearly with the size of the corresponding source-code graph. This is so since every expression tree can at most be marked by one error message. For size of source-code graphs, see Remark 1

5 Correctness and Termination

We now define correctness with respect to errors and termination for graph programs. We give the ideas for proving that `TTC` terminates and is correct w.r.t. errors.

Definition 8 (Correctness and Completeness) A graph program P is *correct with respect to errors* if for every pair $\langle G,H \rangle \in [\![P]\!]$, H is an error graph implies G is not a type-safe source-code graph. If the converse of the implication holds, we say that P is *complete w.r.t. errors*.

Theorem 1 (Correctness) *The graph program* TTC *is correct with respect to errors.*

Proof idea. Errors are only generated by `Recover` and consumes an $=$-edge that should have been consumed by $\texttt{Compare}\downarrow^{+}$, given that TTC were initially dealing with a type-safe source-code graph. A complete proof is given in [AP07].

Fact 1. The graph program TTC is not complete with respect to errors. E.g. `Recover` uses implicit type casts, and thereby avoids generating errors for some non-type-safe graphs. It has not yet been investigated whether or not other counterexamples exist.

Definition 9 (Termination) *Termination* of a graph program is defined inductively on the structure of programs: (1) Every rule p is terminating. (2) For a finite set $\mathscr{S}$ of terminating programs, $\mathscr{S}$ is a terminating program. (3) For terminating programs P and Q, $(P;Q)$ is terminating. Moreover, P^* and $P\downarrow$ is terminating if for every graph G, there is no infinite chain of derivations $G \Rightarrow_P G_1 \Rightarrow_P \ldots$ where $\Rightarrow_P$ denotes the binary relation $[\![P]\!]$ on graphs.

Theorem 2 (Termination) *The graph program* TTC *is terminating.*

Proof idea. `Compare` is applied at least once for every iteration of `TypeCheck` and consumes solid edges that are not generated by the other subprograms. A complete proof is given in [AP07].

6 Conclusions

We considered the template instantiation mechanism in C++ and showed how to write visual rules for type checking and error message generation. We informally described how source code was transformed into source-code graphs and defined type safety for graphs. We transformed source-code graphs into message graphs, a transformation given by the graph program TTC which type checked source-code graphs. The program automatically corrected some errors by implicit type casts. It emitted error messages for type clashes that it could not correct. Proof ideas were given for termination and correctness w.r.t. errors.
Further topics include:

1. Analysis of source-code graphs by generalized graph conditions. Graph properties like "There exists a warning or error node" can be expressed by graph conditions in the sense of [HP05]. It would be interesting to generalize graph conditions so more complex graph properties like "The graph is type safe" becomes expressible.

2. Debugging and a transformation from message graphs to source code. The error messages generated by TTC contained suggestions for remedies. In the double-pushout approach to graph transformation, a central property is the existence of an inverse rule, that when applied reverses the rewrite step of the rule [EEPT06]. In this way, the inverse rule allows for back tracking to a previous graph which can be manipulated to experiment with suggested remedies. The changes are logged in the output graph (message/change graph) and used by a source-code transformer to update the source code.

3. Implementation of the approach. This would include a formalization of the transformation from source code to source-code graphs and an extension of the set of considered template features.

Acknowledgements: This work is supported by the German Research Foundation (DFG) under grant no. HA 2936/2 (Development of Correct Graph Transformation Systems). We thank Annegret Habel for constructive suggestions that improved the paper.

Bibliography

[AG04] D. Abrahams, A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

[AP07] K. Azab, K.-H. Pennemann. Type Checking C++ Template Instantiation (long version). Technical report 04/07, University of Oldenburg, 2007. Available at http://formale-sprachen.informatik.uni-oldenburg.de/%7Eskript/fs-pub/templates_long.pdf.

[CE00] K. Czarnecki, U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer, 2006.

[GPG04] T. Gschwind, M. Pinzger, H. Gall. TUAnalyzer–Analyzing Templates in C++ Code. In *Proc. of WCRE'04*. Pp. 48–57. IEEE Computer Society, 2004.

[HP01] A. Habel, D. Plump. Computational Completeness of Programming Languages Based on Graph Transformation. LNCS 2030, pp. 230–245. Springer, 2001.

[HP05] A. Habel, K.-H. Pennemann. Nested Constraints and Application Conditions for High-Level Structures. LNCS 3393, pp. 293–308. Springer, 2005.

[Jos99] N. M. Josuttis. *The C++ Standard Library: a tutorial and reference*. Addison-Wesley, 1999.

[PMS06] Z. Porkoláb, J. Mihalicza, Á. Sipos. Debugging C++ template metaprograms. In *Proc. of GPCE'06*. Pp. 255–264. ACM, 2006.

[PS04] D. Plump, S. Steinert. Towards Graph Programs for Graph Algorithms. LNCS 3256, pp. 128–143. Springer, 2004.

[Str00] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 2000.

[Str04] B. Stroustrup. Abstraction and the C++ Machine Model. LNCS 3605, pp. 1–13. Springer, 2004.

[Vel98] T. L. Veldhuizen. Arrays in Blitz++. LNCS, pp. 223–230. Springer, 1998.

[VJ02] D. Vandevoorde, N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.