Type Checking C++ Template Instantiation by Graph Programs (long version)

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 lookup 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 add type checking for expressions involving such names. 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 [Str00] 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 separates 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].

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 the definition with the parameterized types 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.

A parameterized type is used to resolve the name `size` in Figure 1. The specialization for the declaration `icon<char>` will not compile since type `char` has no field `size`. If the type `resolution<128>` contains a static field named `size` of an unsigned integer type, then the second specialization will compile.

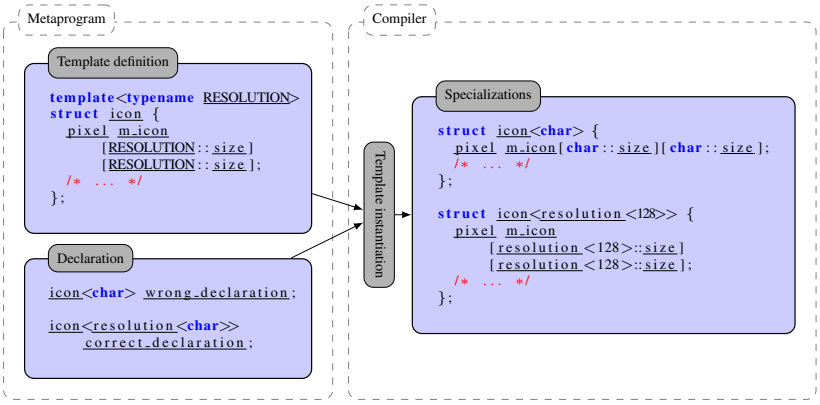


Figure 1: C++ template instantiation.

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.

Graph transformation systems is a well investigated area in theoretical computer science. An overview on the theory and applications is given in the *Handbook on Graph Grammars and Computing by Graph Transformation* [Roz97, EEKR99, EKMR99] and 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, HP07] 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*) performs a subset of the lookup of dependent names and detects type clashes in expressions. `TTC` attempts to solve type clashes by implicit type casts. If such a cast looses precision, a warning message is generated. If no appropriate cast is found, an error messages 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.

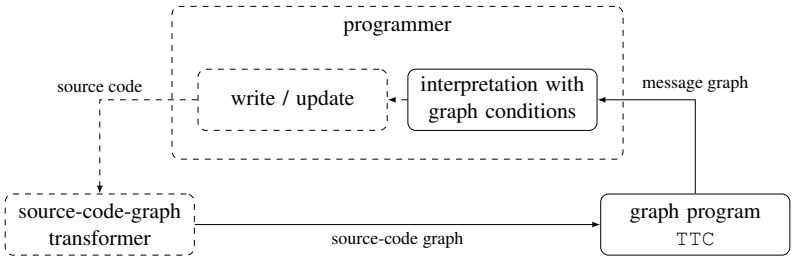


Figure 2: TTC type checks graphs and outputs error messages.

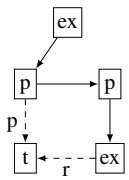
The paper is structured as follows. Graph programs are introduced in Section 2. Section 3 informally describes how C++ source code is transformed into source-code graphs and defines type safety for graphs. In Section 4 we present the graph program TTC for transforming a source-code graph into a message graph. In Section 5 we prove 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. Appendix A collects some lemmas necessary for the proofs on TTC.

2 Graph Programs

In this section, we review graphs, morphisms, conditions, rules, and programs, in the sense of [HP05] and [HP01]. In the following, we consider directed labeled graphs. Labels distinguish different types of nodes and edges and directions model relationships between nodes.

Definition 1 (Graphs) A *label alphabet* $\mathscr{C} = \langle \mathscr{C}_\mathrm{V}, \mathscr{C}_\mathrm{E} \rangle$ consists of two finite sets of node and edge labels. A *graph* over $\mathscr{C}$ is a six-tuple $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{l}_G, \mathrm{m}_G)$ consisting of two finite sets V_G and E_G of *nodes* and *edges*, a *source* and a *target function* for edges $\mathrm{s}_G, \mathrm{t}_G \colon \mathrm{E}_G \to \mathrm{V}_G$, and two *labeling functions* $\mathrm{l}_G \colon \mathrm{V}_G \to \mathscr{C}_\mathrm{V}$ and $\mathrm{m}_G \colon \mathrm{E}_G \to \mathscr{C}_\mathrm{E}$. A graph with an empty set of nodes is *empty* and denoted by $\emptyset$. The set of all graphs over $\mathscr{C}$ is denoted by $\mathscr{G}_\mathscr{C}$. An a-node (edge) in G is an element in V_G (E_G) with label a.

Example 1 *The figure to the right shows a graph over the label alphabet* $\langle \{\mathrm{ex}, \mathrm{p}, \mathrm{t}\}, \{\mathrm{p}_d, \mathrm{r}_d, -_s\} \rangle$*, where d and s denotes a visualization as dashed and solid edges, and* $-$ *is an invisible label. The meaning of the graph, with respect to source code, is explained in Section 3.*



For expressing relationships between graphs we introduce graph morphisms.

Definition 2 (Graph morphisms) A *(graph) morphism* $g \colon G \to H$ consists of two total functions $g_\mathrm{V} \colon \mathrm{V}_G \to \mathrm{V}_H$ and $g_\mathrm{E} \colon \mathrm{E}_G \to \mathrm{E}_H$ that preserve sources, targets, and labels, that is, $\mathrm{s}_H \circ g_\mathrm{E} =$

$g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, $l_H \circ g_V = l_G$, and $m_H \circ g_E = m_G$. It is *injective* if g_V and g_E are injective. The *composition* $h \circ g$ of g with a morphism $h\colon H \to M$ consists of the composed functions $h_V \circ g_V$ and $h_E \circ g_E$.

Example 2 *Let* $G =$ ex op *and* $H =$ ex op. *There exists a unique injective morphism* $G \to H$, *but there is no morphism* $H \to G$, *since the edge in* H *cannot be mapped to an element in* G.

For expressing properties on graphs we use so-called graph conditions. The definition is based on graph morphisms.

Definition 3 (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*.

Example 3 *The graph from Example 1 does not satisfy the condition* $\exists(\emptyset \to$ ex op$)$, *since there exists no morphism from* ex op *to it.*

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 4 (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 $K \to D$, a *direct derivation* consists of two pushouts (1) and (2) where the *match* m and *comatch* m^* are injective. We write a direct derivation $G \Rightarrow_{p,m,m^*} H$. Given a graph G together with an injective match $m : L \to G$, 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 $K \to D$, there is a *direct derivation* $G \Rightarrow_{\hat{p},m,m^*} H$, if $G \Rightarrow_{p,m,m^*} H$, and $m \models \mathrm{ac}$.

Example 4 *Consider the rule* `ResolveSubexpression` *in Figure 3 which deletes and generates two edges. We apply it to the graph from Example 1. We first find the pushout-complement* D *and then the pushout* H.

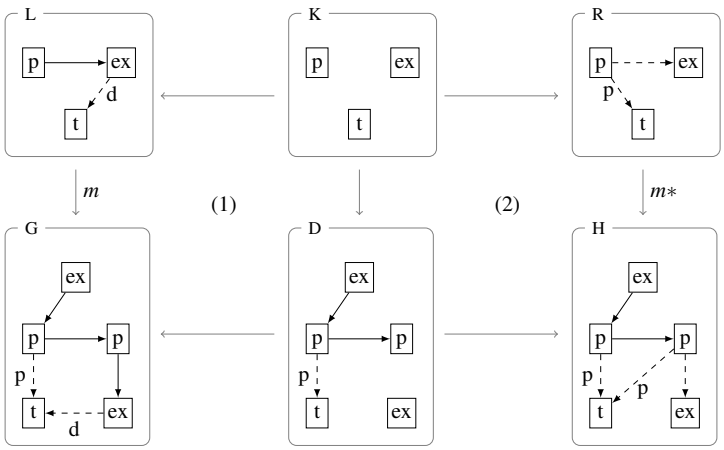


Figure 3: A direct derivation

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

Definition 5 (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 $((P;Q);R)$ are considered as equal; by convention, both can be written as $P;Q;R$. We use $P{\downarrow}^+$ as a shortening of $P;P{\downarrow}$.

Example 5 *For the rule* `ResolveSubexpression` *in Example 4, the iteration as long as possible* `ResolveSubexpression`$\downarrow$ *is a program. Application of that program to G or H, yields the graph H*

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 $\langle L \leftarrow K \rightarrow R\rangle$, 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 the application condition if it is satisfied by any match. To distinguish nodes with the same label, we sometimes print an identifier in the form of "label:id".

Example 6 *Consider the graph program*

$$\texttt{SourceCode} \;=\; \left\{ \begin{array}{c} \texttt{Add},\texttt{Overloading},\texttt{CreateUsage},\texttt{AddCast}, \\ \texttt{CreateExpression1},\texttt{CreateExpression2}, \\ \texttt{AddParameter},\texttt{AddSubexpression}, \end{array} \right\}^*,$$

where `Overloading` = `Overload`;`AddOverload`(p)*;`AddOverload`(r). *The rules are shown in short notation in Figure 4, some of them possess application conditions. The program generates so-called source-code graphs from the empty graph. A source-code graph generated by* `SourceCode` *is shown in Figure 5. The graph is separated 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.*

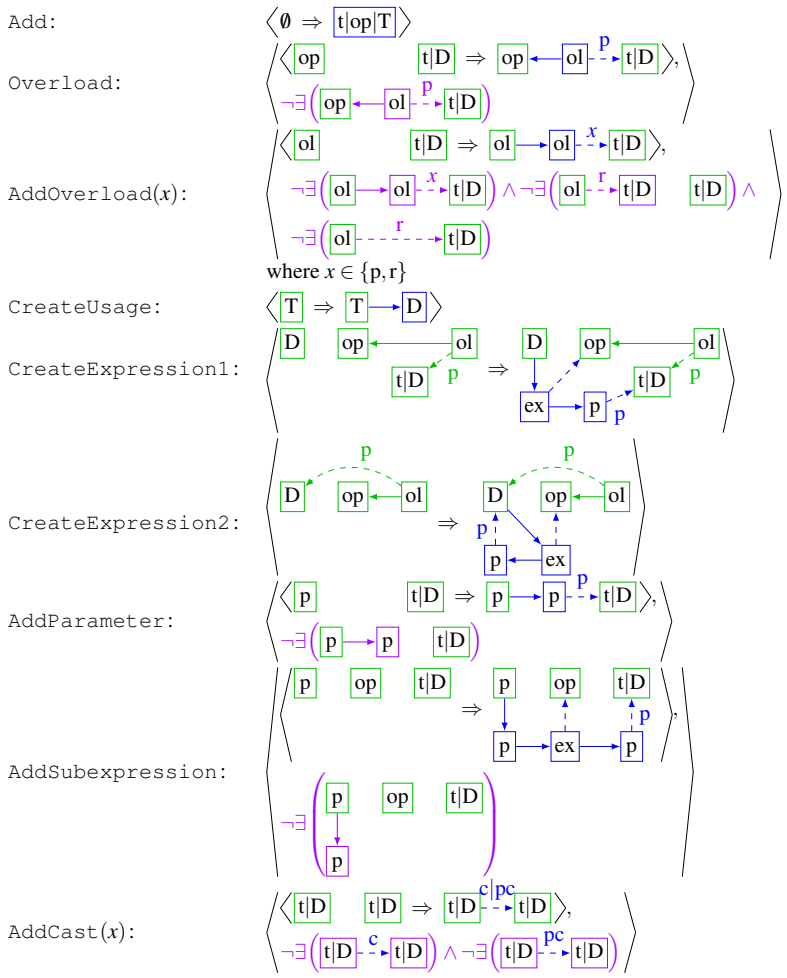


Figure 4: Rules for generating source-code graphs.

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

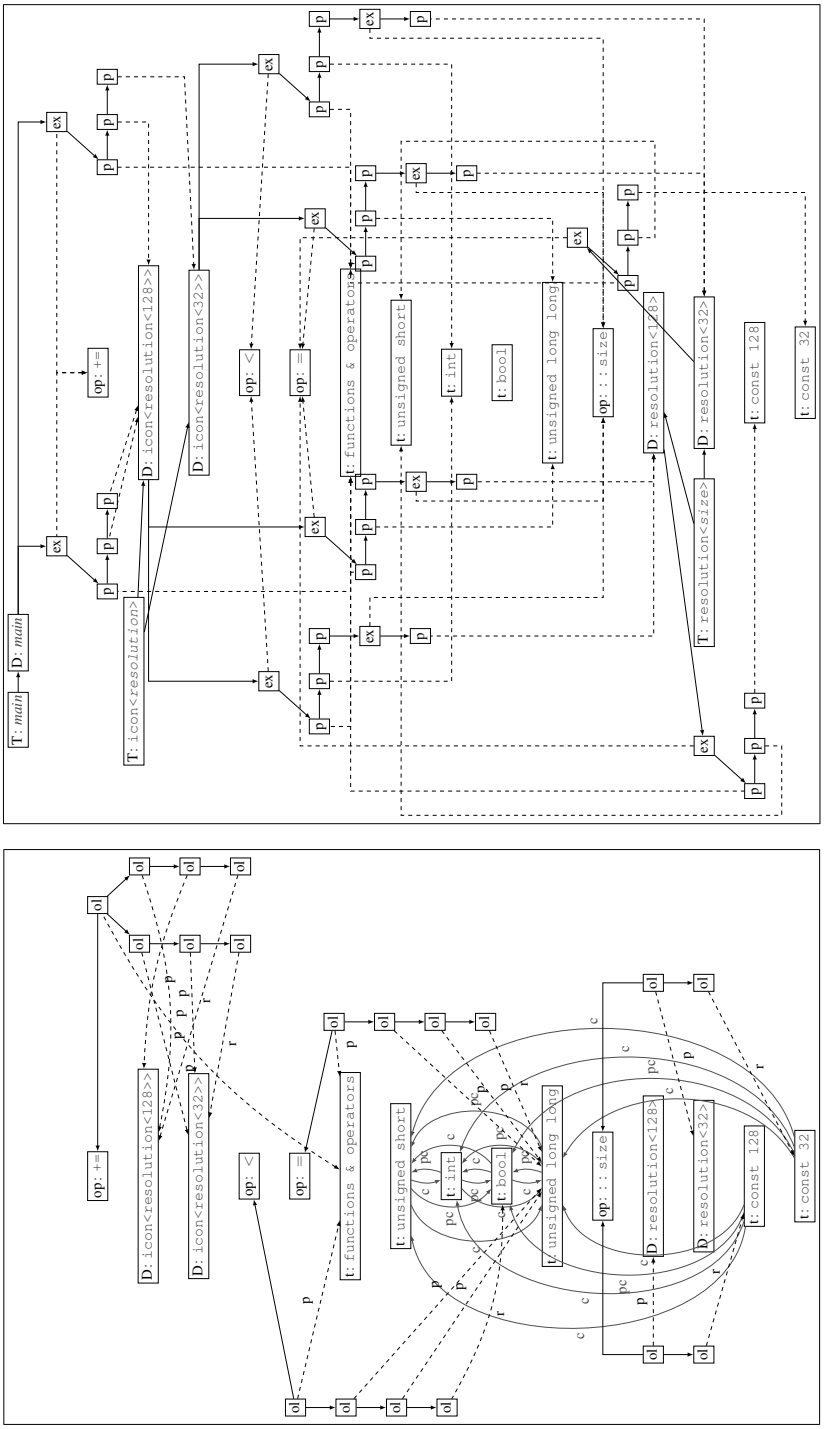


Figure 5: A source-code graph

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 in Table 1.

Table 1: Labels and descriptions.

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.

Example 7 *Consider the source code with two class-type templates, line 1 and 18, in Figure 6. 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 5 shows a source-code graph of the source code in Figure 6.*

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 represent the type information from an expression tree and is modeled by ex- and p-nodes. The root of the tree becomes an ex-node and has an incoming edge from the

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

   struct pixel {
     unsigned char red, green, blue;

     pixel operator+(pixel overlay) {
       pixel result;
       result.red = (red + overlay.red) / 2;
       result.green = (green + overlay.green) / 2;
       result.blue = (blue + overlay.blue) / 2;
       return result;
     }
   };

18 template<typename RESOLUTION>
   struct icon {
20   pixel m_icon[RESOLUTION::size][RESOLUTION::size];

     icon<RESOLUTION>& operator+=(icon<RESOLUTION>& overlay) {
23     for(int i = 0; i < RESOLUTION::size; i++) {
24       for(int j = 0; j < RESOLUTION::size ; j++) {
25         m_icon[i][j] = m_icon[i][j] + overlay.m_icon[i][j];
       }
     }
     return *this;
   }
   };

   #define LARGE_RES resolution<128>
   #define SMALL_RES resolution<32>

   int main() {
     icon<LARGE_RES> pic;
     icon<LARGE_RES> overlay;
     icon<SMALL_RES> low_res;

40   pic += overlay;
41   pic += low_res;

     return 0;
   }
```

Figure 6: Two class-type templates.

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 6 (Expression paths) Given a graph G and a natural number i, an *i-expression path* in G is a path $ex\,p_0 \ldots p_i$, where the head, ex, is an ex-node and $p_0, \ldots, p_i$ 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 8 *An expression tree and its corresponding* 2*-expression path in the context of Example 7 (line 41) is shown in Figure 7.*

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

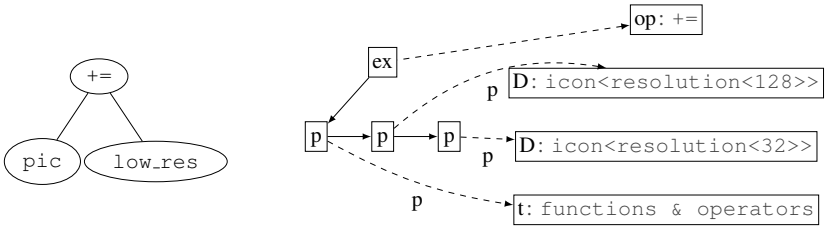


Figure 7: Expression paths represent expression trees.

$n+2$ ol-nodes. The head of that path has an edge to the op-node representing the name `method` 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 7 (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 9 *The overload forest in Figure 8 has one overload tree with two* 2*-overload paths, representing the type signatures of* `icon<resolution<32>>& operator+=(icon<resolution<32>>& overlay)` *and* `icon<resolution<128>>& operator+=(icon<resolution<128>>& overlay).`

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.

Definition 8 (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 (a) t or (b) 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 8 makes the expression path in Figure 7 type safe.

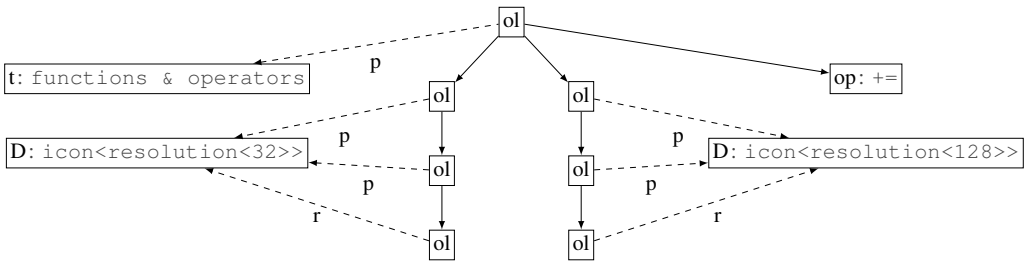


Figure 8: Two overload paths.

4 The Type-Checking Program

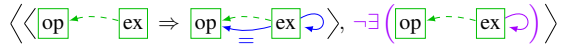
This section describes the graph program `TTC` which performs the name lookup and type checks source-code graphs. The section also shows how message graphs are interpreted with graph conditions.

Definition 9 (`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 9 shows a schematic of how the subprograms of `TTC` interact. Intuitively, `TTC` works as follows: The input is a source-code graph, each expression path is marked and `TypeCheck` is then iterated as long as possible, and the yield is a message graph. Subprograms and rules are described in more detail below.

`MarkExpression`↓ Before the type checking starts, ex-nodes are marked for evaluation by an $=$-edge. To avoid duplicate evaluation, a loop is also placed at the ex-node and checked for in the application condition.

`MarkExpression:` 

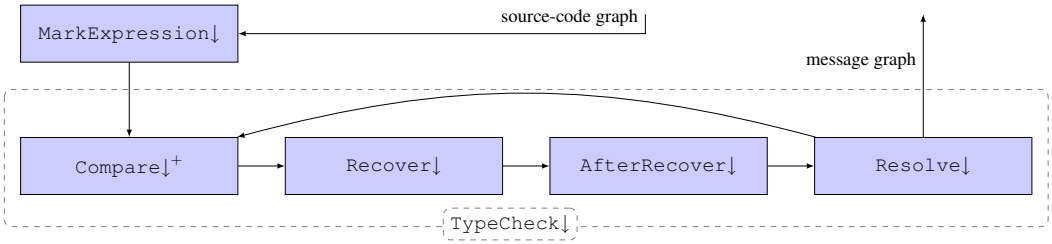


Figure 9: Structure of TTC.

Compare↓+ consists of the rules Lookup, CompareNext, and FindType, see Figure 10. 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 and moves the the =-edge to the first p-node and the head of the overload forest. 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.

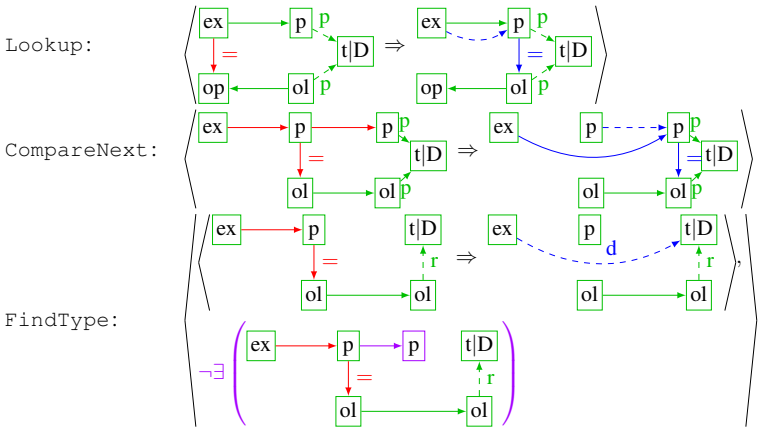


Figure 10: Rules in the program Compare.

Recover↓ consists of the rules Figure 11 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.

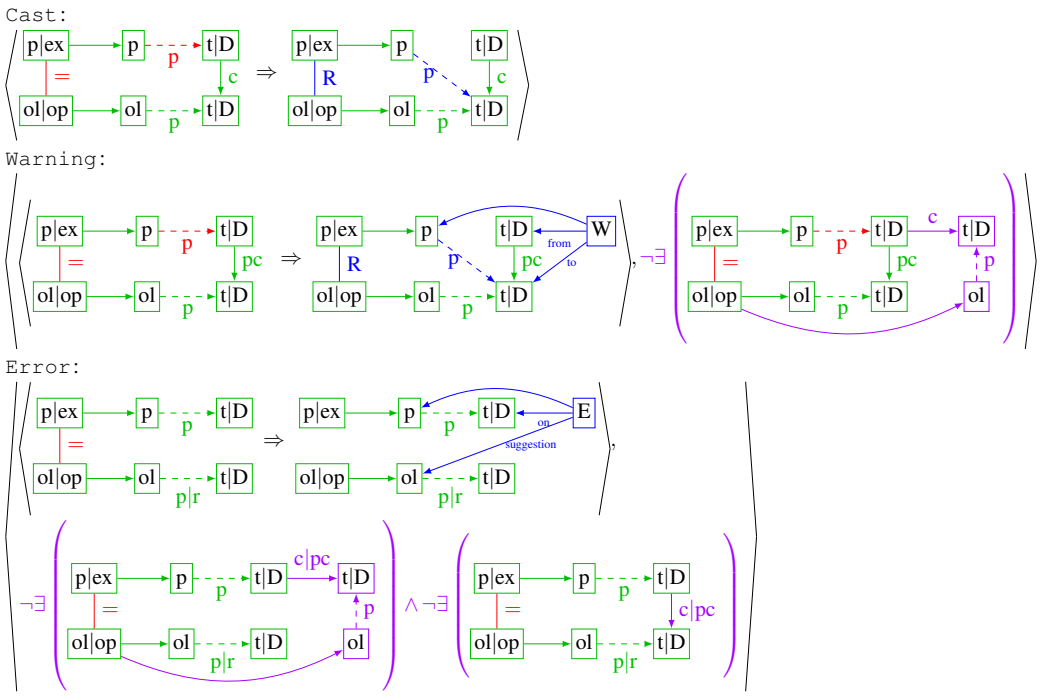
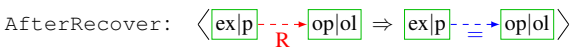


Figure 11: Rules for the program `Recover`.

Remark 1. Since $=$-edges are only generated between nodes of particular labels, we limit the rule schema for rules in `Recover`. This is done by letting $x \in \{\mathrm{p}, \mathrm{ex}\}$ be the label of the node in the top left corner and $f(x)$ be the label of the bottom left node, where f is defined as $f(\mathrm{p}) = \mathrm{ol}$ and $f(\mathrm{ex}) = \mathrm{op}$.

`AfterRecover`$\downarrow$ performs some cleanup work for `Recover`$\downarrow$, generated R-edges are reset to $=$-edges. The limitations described in Remark 1 also applies for `AfterRecover`.



`Resolve`$\downarrow$ consists of three rules: `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.

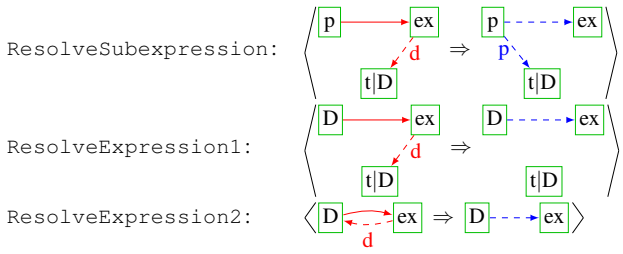


Figure 12: Rules in `Resolve`.

Example 10 *Figure 13 shows the overload tree from Example 9 together with the expression path from Example 8. That expression path has been marked by* `MarkExpression`$\downarrow$. *Figure 14 shows the result of applying the program* `Compare`$\downarrow^+$ *to the graph in Figure 13. Since* `Compare`$\downarrow^+$ *cannot match the third parameter in the expression path to an overload path, the* $=$*-edge remains at the second* p*-node. The graph in Figure 15 is the yield of* `Recover`$\downarrow$ *applied to the graph in Figure 14 as well as the yield of the whole program* `TTC`. *An error message was generated by* `Error` *indicating the location of the type clash and a suggestion for how it can be resolved. Figure 3 shows how* `ResolveSubexpression` *is applied in a nested expression path.*

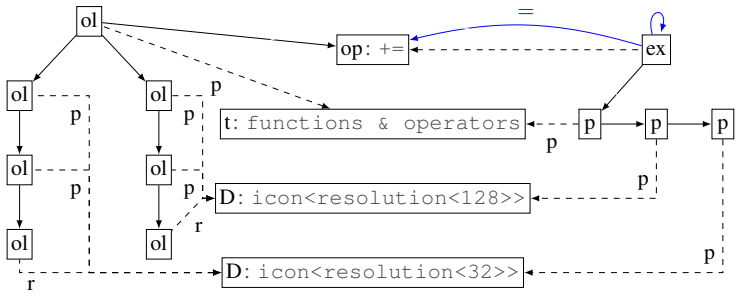


Figure 13: `MarkExpression`$\downarrow$ adds a loop and $=$-edge.

Remark 2. After the termination of `TTC` we use graph conditions 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.

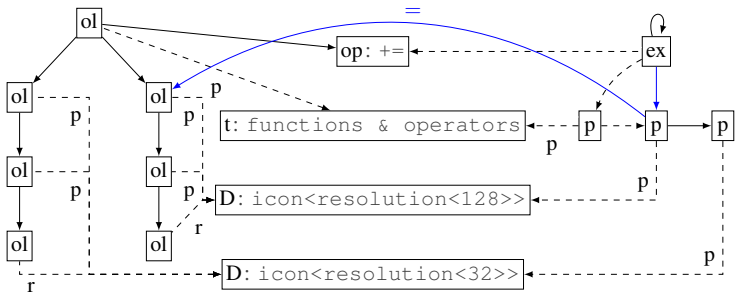


Figure 14: After application of `Compare`$\downarrow^{+}$.

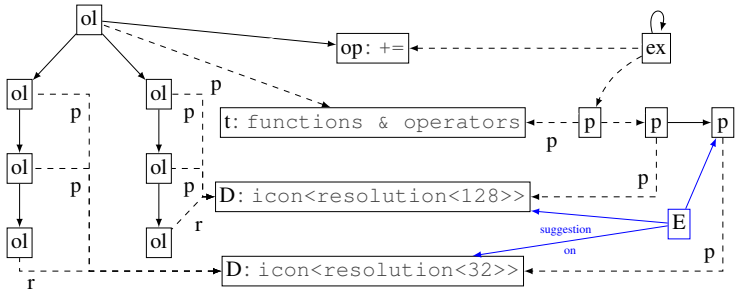


Figure 15: Example yield of `Recover`$\downarrow$.

5 Correctness and Termination

Now we come the the main result of this paper. We show that the graph program for type checking template instantiation is correct with respect to errors, i.e. whenever the application of the program to the input graph yields an error graph, then the input graph had an error, i.e. it was not type safe or not a source-code graph. We also show that `TTC` terminates for all graphs.

Definition 10 (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 with respect to errors*.

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

Proof. Let $\langle G,H \rangle \in [\![\texttt{TTC}]\!]$ and H be an error graph, then there is a graph G_1, such that $\langle G,G_1 \rangle \in [\![\texttt{MarkExpression}\downarrow]\!]$ and $\langle G_1,H \rangle \in [\![\texttt{TypeCheck}\downarrow]\!]$. By Lemma 1 in Appendix A, G_1 is a problem graph (see Definition 13). By Claim 1 in Appendix A, G is not a type-safe source-code graph. □

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.

Next, we define termination for graph programs and show that `TTC` is terminating.

Definition 11 (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.

Remark 3. Termination can be shown by a descending measure function that cannot decrease forever, e.g. by a function $\tau : \mathscr{G}_{\mathscr{C}} \to \mathbb{N}$ such that for every step $G_k \Rightarrow_P G_{k+1}$, $\tau(G_k) > \tau(G_{k+1})$, see [BN98].

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

Proof. Recall that $\texttt{TTC} = \texttt{MarkExpression}\downarrow;\texttt{TypeCheck}\downarrow$. For a graph G, let $\tau_{me}(G) = m$, where m is the number of unmarked ex-nodes in G. Then for every pair of graphs $\langle G, G' \rangle \in [\![\texttt{MarkExpression}]\!]$, $\tau_{me}(G) > \tau_{me}(G')$, i.e. τ_{me} is a termination function. By Lemma 2 in Appendix A, $\texttt{TypeCheck}\downarrow$ terminates. We conclude that `TTC` terminates. □

6 Conclusions

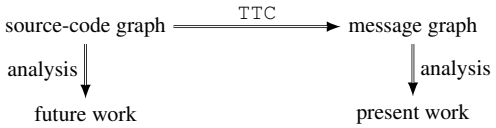
We considered the template instantiation mechanism in C++ and showed how graph programs can detect errors and generate error messages. We informally described how source code was transformed into source-code graphs and defined type safety for graphs.

1. We transformed source-code graphs into message graphs. The transformation was given by the graph program `TTC` which type checked source-code graphs. The program automatically corrects some source-code graphs by implicit type casts. It emits error messages for type clashes that it cannot correct.

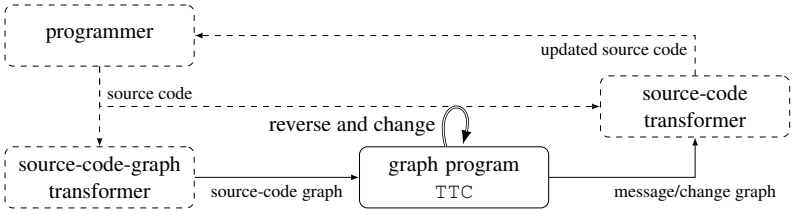
2. We proved that `TTC` terminates and is correct with respect to errors, i.e. whenever the application of the graph program `TTC` yields an error graph, then the input graph had an error – it was not type safe or not a source-code graph.

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, HP07]. It would be interesting to generalize graph conditions so more complex graph properties like "The graph is type safe" becomes expressible. Instead of using the program `TTC` one could directly analyze the source-code graphs, as in the figure below.



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, see the figure below.



3. Extension of the set of considered template features. For practical relevance, it would be important to investigate a more general set of template and other language 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.

[BN98] F. Baader, T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

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

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph*. Volume 2: Applications, Languages and Tools. World Scientific, 1999.

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

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, G. Rozenberg. Handbook of Graph Grammars and Computing by Graph. Volume 3: Concurrency, Parallelism and Distribution. World Scientific, 1999.

[GPG04] T. Gschwind, M. Pinzger, H. Gall. TUAnalyzer–Analyzing Templates in C++ Code. In *11th Working Conference on Reverse Engineering (WCRE'04)*. Pp. 48–57. IEEE Computer Society, 2004.

[HP01] A. Habel, D. Plump. Computational Completeness of Programming Languages Based on Graph Transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*. LNCS 2030, pp. 230–245. Springer, 2001.

[HP05] A. Habel, K.-H. Pennemann. Nested Constraints and Application Conditions for High-Level Structures. In *Formal Methods in Software and System Modeling*. LNCS 3393, pp. 293–308. Springer, 2005.

[HP07] A. Habel, K.-H. Pennemann. Correctness of High-Level Transformation Systems Relative to Nested Conditions. 2007. Submitted.

[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 *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*. Pp. 255–264. ACM, 2006.

[PS04] D. Plump, S. Steinert. Towards Graph Programs for Graph Algorithms. In *Graph Transformations (ICGT'04)*. LNCS 3256, pp. 128–143. Springer, 2004.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph*. Volume 1: Foundations. World Scientific, 1997.

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

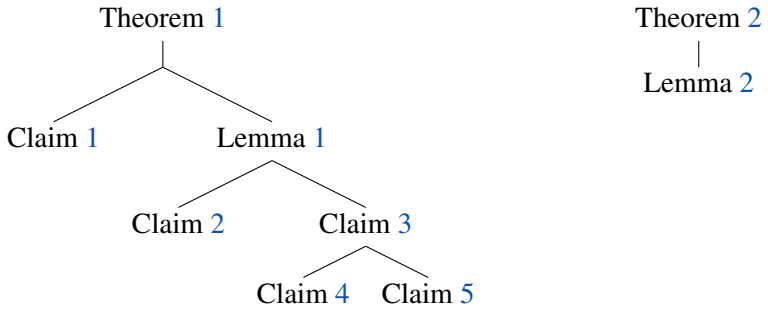
[Str04] B. Stroustrup. Abstraction and the C++ Machine Model. In *Proceedings of the Second International Conference on Embedded Software and Systems*. LNCS 3605, pp. 1–13. Springer, 2004.

[Vel98] T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*. LNCS, pp. 223–230. Springer, 1998.

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

A Proofs

In the following we present the lemmas and claims used in the proofs of Theorem 1 and Theorem 2. The two trees below show how the theorems build upon the lemmas and claims.



For our correctness proof it is useful to introduce a notion of correct $=$- and d-edges. An $=$-edge is used when `TTC` type checks an expression path. If the type checked expression path has an $=$-edge from a position in its path to the equivalent position in the overload path that makes it type safe, then it is a correct $=$-edge. `TTC` uses d-edges to denote the deduced type of an expression path, if the deduced type is the deduced type according to Definition 8, then the d-edge is correct.

Definition 12 (Correct edges) An $=$-edge is correct iff for some natural number i there exists an i-expression path $ex\,p_0 \ldots p_i$ and

- the edge has source ex and target op, where op is the op-node which already has an edge from ex, and there exists a path of solid edges from ex to p_i, or
- there exists an i-overload path $o_0 \ldots o_{i+1}$, such that the overload path makes the expression path type safe and the $=$-edge has source p_k and target o_k, where $0 \leq k \leq i$, and there is a dashed edge from ex to p_k. Furthermore, there exists a path of dashed edges from p_0 to p_k and a path of solid edges from p_k to p_i.

A d-edge is correct if it has source ex and target t, where t is the unique t-node with an r-edge from o_{i+1} (with the overload path as above).

$=$- and d-edges that are not correct are *incorrect*.

With the application of `TTC` to a source-code graph, intermediate graphs are created between rule applications. Certain elements and subgraphs that are (not) found in such intermediate graphs are indications that the input is malformed or not type safe. Such graphs are referred to as problem graphs.

Definition 13 (Problem graphs) A graph is a *problem graph* iff it is not type safe, or contains an incorrect $=$|d-edge, or contains an R-edge or E-node.

Claim 1 (`MarkExpression`$\downarrow$)**.** If $\langle G, H \rangle \in [\![\texttt{MarkExpression}\downarrow]\!]$ and H is a problem graph, then G is not a type-safe source-code graph.

Proof. `MarkExpression` does not generate elements that influence type safety, so if H is not type safe, then G is not type safe. It generates neither R|d-edges nor E-nodes, so if H contains such an element then so does G and is therefore not a source-code graph. By Definition 12, $=$-edges from *ex* to *op* are correct when there is a path of solid edges through the expression path. Since only solid edges are generated between p-nodes by `SourceCode`, the generated $=$-edge is either correct or G is not a source-code graph. It follows that if H contains an incorrect $=$-edge, so does G and is therefore not a source-code graph. This completes the argumentation for the program `MarkExpression`. The statement for `MarkExpression`$\downarrow$ follows from the one for `MarkExpression` and the fact that the rule is non-deleting (of solid edges). □

Lemma 1 (`TypeCheck`$\downarrow$) *If* $\langle G,H\rangle \in [\![\texttt{TypeCheck}\downarrow]\!]$ *and* H *is an error graph, then* G *is a problem graph.*

Proof. Let $\langle G,H\rangle \in [\![\texttt{TypeCheck}\downarrow]\!]$ and H be an error graph. If the program `TypeCheck` was applied to G zero times, then $G \cong H$ and G is an error and problem graph. If the program `TypeCheck` is applied one or more times, $\langle G,H\rangle \in [\![\texttt{TypeCheck}\downarrow^+]\!]$.

`Error` is the only E-node generating rule in `TypeCheck` and there are no E-node deleting rules. If G is not an error graph, then `Error`, and thereby `Recover`, was applied at least once in the derivation sequence from G to H, and

$$\begin{aligned}\langle G,H\rangle \;\in\; [\![\;& (\texttt{Compare}\downarrow^+;\texttt{AfterRecover}\downarrow;\texttt{Resolve}\downarrow)^*;\\ & \texttt{Compare}\downarrow^+;\texttt{Recover}\downarrow^+;\texttt{AfterRecover}\downarrow;\texttt{Resolve}\downarrow;\\ & \texttt{TypeCheck}\downarrow]\!].\end{aligned}$$

Note that the second line in the expression above denotes where `Recover` was first applied. The first line denotes possible applications of `TypeCheck` where `Recover`$\downarrow$ was applied zero times. Only rules in `Recover` generate R-edges and only `AfterRecover` consumes R-edges. If G contains an R-edge, then G is a problem graph, otherwise `AfterRecover` cannot be applied before `Recover` and there exists graphs G_1, G_2, and G_3, such that

$$\begin{aligned}\langle G,G_1\rangle &\in [\![(\texttt{Compare}\downarrow;\texttt{Resolve}\downarrow)^*]\!],\\ \langle G_1,G_2\rangle &\in [\![\texttt{Compare}\downarrow^+]\!],\\ \langle G_2,G_3\rangle &\in [\![\texttt{Recover}]\!], \text{ and}\\ \langle G_3,H\rangle &\in [\![\texttt{Recover}\downarrow;\texttt{AfterRecover}\downarrow;\texttt{Resolve}\downarrow;\texttt{TypeCheck}\downarrow]\!].\end{aligned}$$

By Claim 2 below, G_2 is a problem graph. As a consequence of Claim 3 below, G is a problem graph.

□

Claim 2. If $\langle G,G_1\rangle \in [\![\texttt{Compare}\downarrow^+]\!]$ and $\langle G_1,H\rangle \in [\![\texttt{Recover}]\!]$, then G_1 is not type safe or contains an incorrect $=$-edge.

Proof. By contradiction, assume that G_1 is type safe and contains no incorrect $=$-edge. Since `Recover` is applied to G_1 we know that G_1 contains an $=$-edge. By Definition 12, a correct $=$-edge may exist between nodes

- *ex* and *op*, but then there exists a graph G_2 such that $\langle G_1,G_2\rangle \in [\![\texttt{Lookup}]\!]$, or

- p_k and o_k, where $0 \leq k < i$, but then there exists a graph G_2 such that $\langle G_1, G_2 \rangle \in [\![\texttt{CompareNext}]\!]$, or
- p_i and o_i, but then there exists a graph G_2 such that $\langle G_1, G_2 \rangle \in [\![\texttt{FindType}]\!]$,

and therefore $\langle G, G_1 \rangle \notin [\![\texttt{Compare}\downarrow^+]\!]$, which is a contradiction. Consequently G_1 is not type safe or contains an incorrect $=$-edge. □

Claim 3. If $\langle G, H \rangle \in [\![(\texttt{Compare}\downarrow;\texttt{Resolve}\downarrow)^*;\texttt{Compare}\downarrow^+]\!]$ and G is not a problem graph, then H is not a problem graph.

Proof. Let $\langle G_1, H_1 \rangle \in [\![\texttt{Compare}]\!]$, $\langle G_2, H_2 \rangle \in [\![\texttt{Resolve}]\!]$, and G_1 and G_2 not be problem graphs. By Claim 4 below, H_1 is not a problem graph. By Claim 5 below, H_2 is not a problem graph. The statement for the program $(\texttt{Compare}\downarrow;\texttt{Resolve}\downarrow)^*;\texttt{Compare}\downarrow^+$ follows immediately. □

Claim 4. If $\langle G, H \rangle \in [\![\texttt{Compare}]\!]$ and G is not a problem graph, then H is not a problem graph.

Proof. Let $\langle G, H \rangle \in [\![\texttt{Compare}]\!]$ and G not be a problem graph. Recall $\texttt{Compare} = \{\texttt{Lookup}, \texttt{CompareNext}, \texttt{FindType}\}$. If the rule:

- `Lookup` is applied at a correct $=$-edge between ex and op, then the generated $=$-edge between p_0 and o_0 is correct by Definition 12, or there exists another overload tree with a root equivalent to o_0, and then G does not contain an overload forest (see Definition 7) and is therefore not type safe, which is a contradiction. Elements generated by `Lookup` do not influence type safety, hence H is type safe. `Lookup` does not generate d-edges, or elements critical to correct d-edges, so all d-edges in H are correct.
- `CompareNext` is applied at a correct $=$-edge from a node p_k in an i-expression path to node o_k in an i-overload path, where $0 \leq k \leq i-1$, then the generated $=$-edge between p_{k+1} and o_{k+1} is correct by Definition 12, or there exists a sibling to o_0 forbidden by Requirement 3 in Definition 7 and G is therefore not type safe, which is a contradiction. Elements generated by `Lookup` do not influence type safety, hence H is type safe. `CompareNext` does not generate d-edges or elements critical to correct d-edges, so all d-edges in H are correct.
- `FindType` is applied at a correct $=$-edge from p_i to o_i, then the generated d-edge between ex and t is correct by Definition 12, or o_k has two children with r-edges, which is forbidden for overload trees (see Definition 7, Requirement 4) and G can therefore not be type safe, which is a contradiction. Elements generated by `Lookup` do not influence type safety, hence H is type safe. `Lookup` does not generate $=$-edges nor elements that influence correct $=$-edges, so all $=$-edges in H are correct.

Neither rule generates R-edges or E-nodes, so H contains no R-edges or E-nodes. □

Claim 5. If $\langle G, H \rangle \in [\![\texttt{Resolve}]\!]$ and G is not a problem graph, then H is not a problem graph.

Proof. Let $\langle G,H\rangle \in [\![\texttt{Resolve}]\!]$ and G be type safe and contain no incorrect =- or d-edges. Let

$$\texttt{Resolve} = \left\{ \begin{array}{c} \texttt{ResolveExpression1}, \texttt{ResolveExpression2}, \\ \texttt{ResolveSubexpression} \end{array} \right\}.$$

If the rule:

- `ResolveSubexpression` is applied at a correct d-edge, the generated p-edge to t do not influence type safety, see Definition 8, Requirement 2.a and Requirement 2.b are interchangeable. The rule `ResolveSubexpression` does not delete elements critical to correct =- or d-edges, so the =- and d-edges in H are correct. Neither does the rule generate R-edges or E-nodes, so H contains no R-edges or E-nodes.

- `ResolveExpression1` is applied, the statement for H is true. No elements are deleted that influence type safety or the correctness of =- or d-edges, and R-edges and E-nodes are not generated.

- `ResolveExpression2` is applied, the statement for H is true. Same argumentation as for `ResolveExpression1`.

□

Lemma 2 (Termination of `TypeCheck`↓) *The graph program* `TypeCheck`↓ *is terminating.*

Proof. Recall that

$$\texttt{TypeCheck} = \texttt{Compare}\!\downarrow^{+};\texttt{Recover}\!\downarrow;\texttt{AfterRecover}\!\downarrow;\texttt{Resolve}\!\downarrow, \text{and}$$

$$\texttt{Compare} = \{\texttt{Lookup}, \texttt{CompareNext}, \texttt{FindType}\}.$$

1. For arbitrary graph G, let $\tau_{co}(G) = 2m+n+o$, where m is the number of =-edges going from an ex-node to an op-node, n is the number of solid edges between two p-nodes, o is the number of =-edges between a p-node and an ol-node in G. If $\langle G,G'\rangle \in [\![\texttt{Lookup}]\!]$, then $\tau_{co}(G) > \tau_{co}(G') = 2(m-1)+n+o+1$. If $\langle G,G'\rangle \in [\![\texttt{CompareNext}]\!]$, then $\tau_{co}(G) > \tau_{co}(G') = 2m+n-1+o$. If $\langle G,G'\rangle \in [\![\texttt{FindType}]\!]$, then $\tau_{co}(G) > \tau_{co}(G') = 2m+n+o-1$. Hence, τ_{co} is a termination function for `Compare`↓ and `Compare`↓ is terminating.

2. Rules in `Recover` consume =-edges. For a graph G, let $\tau_{rec}(G) = m+o$, where m and o have the same meaning as above. If $\langle G,G'\rangle \in [\![\texttt{Recover}]\!]$, then $\tau_{rec}(G) > \tau_{rec}(G')$, i.e. τ_{rec} is a termination function for `Recover`↓. Hence, `Recover`↓ terminates.

3. `AfterRecover` consumes R-edges. For any graph G, let $\tau_{ar}(G) = m'+o'$, where m' and o' are the number of R-edges in G from an ex-node to op-node and p-node to ol-node, respectively. If $\langle G,G'\rangle \in [\![\texttt{AfterRecover}]\!]$, then $\tau_{ar}(G) > \tau_{ar}(G')$, i.e. τ_{ar} is a termination function for `AfterRecover`↓ and `AfterRecover`↓ terminates.

4. All rules in `Resolve` consume a d-edge going from an ex-node to a t- or D-node. For a graph G, let $\tau_{res}(G) = q$, where q is the number of d-edges in G. If $\langle G, G' \rangle \in [\![\texttt{Resolve}]\!]$, then $\tau_{res}(G) > \tau_{res}(G')$. τ_{res} is a termination function for `Resolve`$\downarrow$ and `Resolve`$\downarrow$ terminates.

By 1.-4., `TypeCheck` terminates. For a graph G, let $\tau_{tc}(G) = 2m + n + o + 2m' + o'$, where m, n, o, m', and o' have the same meaning as above. If $\langle G, G' \rangle \in [\![\texttt{TypeCheck}]\!]$, then there are graphs G_1, G_2, and G_3, such that

$$\begin{array}{rcl} \langle G, G_1 \rangle & \in & [\![\texttt{Compare}\downarrow^+]\!], \\ \langle G_1, G_2 \rangle & \in & [\![\texttt{Recover}\downarrow]\!], \\ \langle G_2, G_3 \rangle & \in & [\![\texttt{AfterRecover}\downarrow]\!], \text{ and} \\ \langle G_3, G' \rangle & \in & [\![\texttt{Resolve}\downarrow]\!]. \end{array}$$

We know $\tau_{co}(G) > \tau_{co}(G_1)$ and no rule in `Compare` generates R-edges, so $\tau_{tc}(G) > \tau_{tc}(G_1)$. Rules in `Recover` consumes $=$-edges and might generate R-edges, yielding $\tau_{tc}(G_1) \geq \tau_{tc}(G_2)$. `AfterRecover` turns a R-edge into a $=$-edge and therefore $\tau_{tc}(G_2) = \tau_{tc}(G_3)$. `Resolve` does not influence the termination function, so $\tau_{tc}(G_3) = \tau_{tc}(G')$. We conclude $\tau_{tc}(G) > \tau_{tc}(G')$, i.e. τ_{tc} is a termination function for `TypeCheck`$\downarrow$. Hence `TypeCheck`$\downarrow$ terminates.

□