A Formalism for C++

N1885=05-0145

Gabriel Dos Reis
Department of Computer Science
Texas A&M University
College station, TX-77843
gdr@cs.tamu.edu

Bjarne Stroustrup
Department of Computer Science
Texas A&M University
College station, TX-77843
and AT&T Labs – Research
bs@cs.tamu.edu

Abstract

In this paper, we develop a general formalism for describing the C++ programming language, and regular enough to cope with proposed extensions (such as *concepts*) for C++0x that affect its type system. Concepts are a mechanism for checking template arguments currently being developed to help cope with the massive use of templates in modern C++. The main challenges in developing a formalism for C++ are scoping, overriding, overloading, templates, specialization, and the C heritage exposed in the built-in types. Here, we primarily focus on templates and overloading.

1 Introduction

Over the last two decades, the C++ programming language [ISO98, ISO03, Str00] has been adopted in wide and diverse application areas [Str]. Beside traditional applications of general purpose programming languages, it is being used in embedded systems, safety-critical missions, air-plane and air-traffic controls, space explorations, etc. Consequently, the demand for static analysis and advanced semantics-based transformations of C++ programs is becoming pressing. The language supports various mainstream programming styles, all based on static type checking. In particular, it directly supports generic programming through the notion of *template* for capturing commonalities in abstractions while retaining optimal efficiency. It directly supports object-oriented programming through *inheritance* and *virtual functions*. It also offers function *overloading*. Various extension proposals for C++0x, the next version of ISO Standard C++, are under active discussions. The major ones affect its type system.

To analyse and precisely specify existing and proposed features, we need a formal framework for describing the static semantics of ISO C++. Such a formalism must be general enough to accommodate the many proposed language extensions and be a convenient tool for their evaluations. We are building such a framework as part of *The Pivot* project, an infrastructure for C++ program analysis and semantics-based transformations. One early aim is to provide theoretical and experimental support *concepts*, a typing mechanism for template arguments; basically, a concept is the type of a type. In this paper, we will focus on ISO C++'s type and template system.

1.1 Our contributions

Our main contribution is a formalism capable of describing the ISO C++ semantics (especially its template system) and likely new features, such as concepts for C++0x [DRS05]. In particular, the formalism allows us to clearly distinguish concepts from Haskell type classes [WB89, HHPJW96]. Furthermore, we use the formalism to clarify the distinctions and interactions between *overloading*, *template partial specialization*, and *overriding*; three distinctive features that have often been confused in the literature. We developed this formalism side-by-side with a complete, minimal representation of C++ code so we are confident that the formalism has direct practical implications for the specification of C++ analysis and transformation tools. As an indication of the inherent simplicity of the formalism, we can mention that our library for efficiently representing all of ISO C++ according to the rules defined by the formalism is just 2,500 lines of C++ source code (including all memory management and type unification).

1.2 Outline

This paper is structured as follows. The next section provides a background on the ISO C++ programming language. In §3 we describe an internal form that is suitable for use in compilers (type checking, code generation) and more generally for semantics-based program transformations. Then we move on the description of an external form (§4), followed by a discussion of the template system (§5). We give in §6 a brief outline of application of the formalism developed in this paper to the specification of a type system for templates. §7 discusses related works, and we conclude in §8 with directions for future work.

2 Background

A formal description of the static semantics of C++ is a challenging task. Part of the challenge comes from the fact that C++ is based on the C programming language, whose static semantics has resisted formal approaches [Set80, GH93]. An-

other part of the challenge has its root in the language design principle that C++ should not be just a collection of neat features. That is, C++ language functionalities strongly interact with each other to sustain sound programming styles. The features are designed to be used in combination. Consequently, it is hard to describe a feature completely, in isolation from other features. For example, function templates can be overloaded and a class template can have member functions that override virtual functions from base class templates, etc. supporting a wide range of useful, sound programming styles and techniques. In particular, the Standard Template Library [SL94] critically relies on templates and overloading. Consider the function template declarations:

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first,
          RandomAccessIterator last);

template<class Container>
void sort(Container& c);
```

The first declaration is quite standard [SL94, Aus98, Str00, ISO03]. Its classical meaning is: given a sequence delimited by the interval $[\mathtt{p},\mathtt{q})$, the call `sort(p, q)` will perform an in-place sorting assuming `p` and `q` have the properties of *random access iterators*. The second declaration is less standard, but is part of some library extensions and has been used as an example [Str00]. It is that of a sorting function that takes a whole container argument instead of parts of it. From standard C++ point of view, the function template `sort()` is *overloaded*. Note that, contrary to systems like type classes [WB89, Tha94, HHPJW96, PJ03], there is no requirement for the declaration of a global repository and explicit registration of members of that repository forming the overload set. It should also be noted that members of the overload set need not have the same number of arguments, nor have any generic instance relationship. Furthermore, unlike the language discussed in the work of Peter Stuckey and Martin Sulzmann [SS02] no keyword `overload` is needed to introduce a function declaration as overloaded. In fact, the keyword `overload` was required in early (pre-release, internal to AT&T) versions of C++, but experience and and experiments were quite negative. The conclusion was that such schemes do not scale to programming in the large as they seriously inhibit composability of independently developed libraries [Str94, §11.2.4].

The IOStream [LK00] component relies on overloading, templates and virtual functions (thus overriding) to offer an extensible, type safe, efficient input/output library. For example, consider the fragment

```
double x = 3.9;
//...
cout << "x = " << x << '\n';
```

The references to the insertion operator («) are to a function that is overloaded for various types, including character strings, double precision floating point types, character types. Overload resolution, in C++, considers only the *static* types of the operands. That mechanism supports extensions of the IOStream component to user-defined types and input/output schemes. Furthermore, the input and output streams, as well as the stream buffers and localization helper classes are statically parameterized by the character type manipulated by the streams. This makes it possible for users to instantiate the framework with their own character types, such as Unicode characters. Finally, the stream classes and helper classes use overriding of virtual functions to provide hooks for dynamic customization points for input and output processing. The IOStream component is just an example of many successful C++ libraries that combine overloading, templates and overriding to provide efficient, flexible, type safe and extensible framework.

The rest of this paper focuses on a formal description of two of the key functionalities of the ISO standard C++ programming language: overloading and templates (including partial specialization).

3 The internal language

C++ is an explicitly typed language. However, the source language requires annotations only on declarations. Template instantiations are usually implicit and overloading is resolved by matching argument list with the types of visible function declarations. A given call may involve template-argument deduction (§5.1), followed by function overloading resolution (§4.2), which in turn is followed by template instantiation. The outcome is a fully typed intrinsic language. This section describes the internal form of C++. We start with a description of the core of its type system, necessary to discuss the main ideas behind templates and overloading.

3.1 The type system

The abstract syntax for types is given in Figure 1 and judgment rules for type expressions are displayed in Figure 2. Sentences of the form $\Gamma \overset{\text{type}}{\vdash} t \in T$ means "the type expression t has property T." Similarly, sentence like $\Gamma \overset{\text{decl}}{\vdash} x \in X$ means "the declaration of x has property X".

Type constants include built-in types such as `void`, `int`, etc., as well as user-defined types (classes, unions, enums, etc.) For generality and uniformity, we also include "generalized types" such as *class*, *enum*, *union* and *namespace*. They describe the type of user-defined types. For instance, in the fragment

```
namespace ast {
  enum token {
    // ...
  };

  class expr {
    // ...
  };
}
```

the namespace `ast` is considered a user-defined type, with type *namespace*. Similarly `ast::token` and `ast::expr` are user-defined types with types *enum* and *class*, respectively.

Type variables are generally introduced as type template-parameters. We will see in §5.5 that they can also arise as synthesized types in abstract typing of templates definitions. A reference type is constructed with $ref(-)$, and a pointer type is constructed with $ptr(-)$.

Types constructed with the keywords `const` and `volatile`

$$
\begin{array}{lll}
\tau ::= & & \textit{types} \\
& \mathtt{t} & \textit{type constants} \\
& \mid \alpha & \textit{type variables} \\
& \mid \mathit{ref}(\tau) & \textit{reference types} \\
& \mid \mathit{ptr}(\tau) & \textit{pointer types} \\
& \mid \mathit{const}(\tau) & \texttt{const}\textit{-qualified types} \\
& \mid \mathit{volatile}(\tau) & \texttt{volatile}\textit{-qualified types} \\
& \mid \mathit{array}(\tau, c) & \textit{array types with known bound} \\
& \mid \mathit{array}(\tau, \varepsilon) & \textit{array types with unknown bound} \\
& \mid (\tau_1, \cdots, \tau_n) \to \tau & \textit{function types} \\
& \mid \Pi[\![\vec{p} : \mathbb{T}^n]\!] \twoheadrightarrow \tau & \textit{type template} \\
& \mid \chi_{[\gamma_1, \cdots, \gamma_n]} & \textit{type template instantiations} \\
\mathbb{T} ::= & & \textit{generalized types} \\
& \tau & \textit{ordinary type} \\
& \mid \natural & \textit{type of ordinary type} \\
& \mid \mathbb{T}_1 \times \cdots \times \mathbb{T}_n \twoheadrightarrow \mathbb{T} & \textit{template type} \\
\gamma ::= & & \textit{compile-time entities} \\
& c & \textit{constant expressions} \\
& \mid \tau & \textit{types} \\
& \mid \chi & \textit{named type templates}
\end{array}
$$

Figure 1: Abstract syntax of the type system

$$
\frac{}{\Gamma \overset{\text{type}}{\vdash} \mathscr{T} \in \natural}
$$

$$
\frac{}{\Gamma \overset{\text{type}}{\vdash} \mathtt{t} \in \mathscr{T}} \quad \textit{where} \quad \mathtt{t} ::= \mathtt{void} \mid \mathtt{bool} \mid \mathtt{int} \mid \mathtt{double} \mid \ldots
$$

$$
\frac{\Gamma \overset{\text{type}}{\vdash} \tau \in \mathscr{O} \cup \mathscr{F}}{\Gamma \overset{\text{type}}{\vdash} \mathit{ref}(\tau) \in \mathscr{R}} \qquad \frac{\Gamma \overset{\text{type}}{\vdash} \tau \in \mathscr{T}^{\times}}{\Gamma \overset{\text{type}}{\vdash} \mathit{ptr}(\tau) \in \mathscr{O}}
$$

$$
\frac{\Gamma \overset{\text{type}}{\vdash} \tau \in \mathscr{T}^{\times}}{\Gamma \overset{\text{type}}{\vdash} \mathit{const}(\tau) \in \mathscr{T}^{\times}} \qquad \frac{\Gamma \overset{\text{type}}{\vdash} \tau \in \mathscr{T}^{\times}}{\Gamma \overset{\text{type}}{\vdash} \mathit{volatile}(\tau) \in \mathscr{T}^{\times}}
$$

$$
\frac{\Gamma \overset{\text{type}}{\vdash} \tau \in \mathscr{O}}{\Gamma \overset{\text{type}}{\vdash} \mathit{array}(\tau, \_) \in \mathscr{O}} \qquad \frac{\forall i \; \Gamma \overset{\text{type}}{\vdash} \tau_i \in \mathscr{P} \quad \tau \in \mathscr{T}^{+}}{\Gamma \overset{\text{type}}{\vdash} (\tau_1, \cdots, \tau_n) \to \tau \in \mathscr{F}}
$$

$$
\frac{\Gamma \overset{\text{decl}}{\vdash} \upsilon \in \{\mathit{class}, \mathit{union}, \mathit{enum}\}}{\Gamma \overset{\text{type}}{\vdash} \upsilon \in \mathscr{O}} \qquad \frac{\Gamma \overset{\text{decl}}{\vdash} \alpha \in \mathit{typename}}{\Gamma \overset{\text{type}}{\vdash} \alpha \in \mathscr{T}}
$$

$$
\frac{\Gamma \overset{\text{decl}}{\vdash} \chi \in \mathbb{T}_1 \times \cdots \times \mathbb{T}_n \twoheadrightarrow \tau \qquad \forall i \; \gamma_i \in \mathbb{T}_i, \; \tau \in \mathbb{T}}{\Gamma \overset{\text{type}}{\vdash} \chi_{[\gamma_1, \cdots, \gamma_n]} \in \tau}
$$

Figure 2: Typing judgment for type expressions

are represented as $\mathit{const}(-)$ and $\mathit{volatile}(-)$. Array types, constructed with array, may have unknown bounds or must have bounds that are integral constant expressions. Functions declared to take $n \geq 0$ arguments and with return type τ have types denoted by $(\tau_1, \cdots, \tau_n) \to \tau$. When $n = 0$, that is when the function takes no arguments, the notation reduces to $() \to \tau$. A type template is a mapping from compile-time entities (type, values, named templates) to types. Types can also be constructed as instantiations of type templates with arguments that are *compile-time entities*. Type templates are structured by the form of their template-parameters (type, non-type and template). Finally, the collection of template-arguments consists of C++ constant expressions, types and named type templates. The structure of constant expressions is precisely defined by the ISO standard rules [ISO03, §5.19]. For the purpose of this paper, it would suffice to say that they are the sort of values that can be computed at translation-time as part of type-checking.

Types in $\mathbb{T}$ are rather "enthusiastic" extensions of the ISO C++ type system in the sense that we tolerate some types expression, e.g. pointer to reference or array of references, that are considered invalid by ISO C++ rules. This allows us to provide a significantly smaller and more regular description. However, to remain close to those rules, we use $\mathscr{T}$ to denote the collection of ISO C++ well-formed types. All types in $\mathscr{T}$ have kind $\natural$. They consist of the type constant `void` and the three categories of types:

1. value and object types $\mathscr{O}$;
2. function types $\mathscr{F}$;
3. reference types $\mathscr{R}$;

We single out the sub-collection $\mathscr{T}^{+} = \mathscr{T} \setminus \mathscr{F}$ composed of all well-formed types except function types. Types in $\mathscr{T}^{+}$ are those that can serve as function return type. Similarly, the collection $\mathscr{P} = \mathscr{T}^{+} \setminus \{\mathtt{void}\}$ is that of function parameter types. This is a slight extension of ISO C++ types that can be used as function return type and parameter types. Indeed, they include array types. The traditional and standard rules say that the return type of a function cannot be an array type. However, there is no fundamental intrinsic reasons why that should be the case; that is an instance of historical accidents. Similarly, the standard rules say that a function parameter cannot have an array type. Or rather, if it is declared to have an array type then it is implicitly adjusted to a pointer type. This particular aspect is no source of departure from ISO C++ standard semantics because it is expected that where enforcing ISO C++ rules, the program translation will make the appropriate adjustment during conversion from the external to the internal forms, as is done in C++ compilers. However, the formalism is capable to dealing with such extensions did the need arise. The result is a type system that is more regular, yet not compromising C++ standard semantics. We also use the sub-collection $\mathscr{T}^{\times} = \mathscr{T} \setminus \mathscr{R}$, which is the collection of types from which an ISO C++ pointer type can be constructed.

The collection $\mathscr{O}$ consists of types that describes *objects*, e.g. representation of values in memory. At the exception of `void`, all built-in types are object types. Classes, enums and unions are also object types. Figure 2 displays the semantics of the types in $\mathscr{T}$. A reference type can be constructed only out of an object or function type. A pointer type can be constructed only out of `void`, or an object type, or a function type. In particular, there is no pointer to reference type or pointer to namespaces (though the latter may be considered in the specification of some form of module systems).

Cv-qualification, e.g. construction with *const* or *volatile*, preserves categories. That is, a cv-qualification of an object type is an object type, and a cv-qualification of a function type is a function type. There is no notion of cv-qualification of reference type. The standard rules do not allow cv-qualification of function types, except in very limited occasions. However, it turns out that the general precise formulation of template-argument deduction process becomes quite complicated. Consequently, we have decided to include cv-qualification in our framework and reject programs that *effectively* use them in way that contradicts ISO C++ semantics.

Array types are object types, and are constructed only out of object types. A function type may have any well-formed type as return type, and its parameter-type list consists of either object types or reference types. A symbol declared to name a class, a union or an enumeration designates an object type. In a template declaration, a name introduced with the keyword `typename` designates a type. A symbol introduced to name a type template is interpreted as a type template. Finally, a type template instantiated with the appropriate template arguments has a result which is of the category indicated in its signature.

3.2 Declarations

As a general C++ rule, a name can be used to form expressions only if it has a prior visible declaration stating its type. Declarations are collected in look-up tables (e.g. *scopes*) that record type assumptions associated with names. Declarations are categorized into various sorts, depending on the type and the scope where they appear. There are six sorts of scopes:

1. *namesapce-scope* associated with namespaces;
2. *class-scope* associated with classes;
3. *function-scope* a curiosity associated with labels;
4. *template-scope*, associated with template-parameters;
5. *prototype-scope*, associated with function parameters in function declarations (not definitions);
6. *local-scope* associated with blocks.

The abstract syntax and the typing judgment rules are displayed in Figure 3 and Figure 4, respectively. The general principle is that a declaration is a tuple $n : (\Gamma, \tau, \sigma)$ with an optional initializer where

- Γ is the scope into which the name n is introduced;
- τ is the type of the declaration;
- and σ is the set of specifiers.

For example consider:

```
int abs(int);

class Var {
public:
   const string& name() const;
   const Type& type() const;
   const Expr& initializer() const;
   //...
};
```

$d ::=$	*declarations*
$n : (\Gamma, \tau, \sigma)$	*declaration without initializer*
$n : (\Gamma, \tau, \sigma) \leftarrow e$	*declaration with initializer*
$\sigma ::=$	*specifiers*
`auto` | `register`	
| `static` | `extern`	
| `mutable` | `export`	
| `inline` | `virtual`	
| `explicit` | `pure`	
| `public` | `private`	
| `protected`	

Figure 3: Abstract syntax of C++ declarations

```
template<typename T, int N> struct buffer {
   T data[N];
};
```

The first declaration has name `abs` and type $(\texttt{int}) \to \texttt{int}$ with no initializer. The class declaration for `Var` has type *class* and the initializer is the class-body. The third declaration has name `buffer`, has type $\natural \times \texttt{int} \twoheadrightarrow$ *class* and the class-body as initializer.

The operation of retrieving the set of declarations for a name n in a given scope Γ is called *name lookup*, and will be denoted $\mathcal{L}(\Gamma, n)$. We will assume that operation to be given for the purpose of this paper. The reader interested in the exact details of ISO C++ name lookup rules might consult [ISO03, §3.4] for the standard specifications. Name lookup applies uniformly to all names.

A variable is a declaration that binds a name to an object type at a non-class scope. A reference is a binding of a name to a reference type at a non-class scope. Declarations appearing at class-scope are member declarations. A declaration of a name with object type or reference type at a class-scope is a data-member (or field) provided it has no `static` specifier. A function declaration binds a name to a function type. A declaration that appears at template-scope or prototype-scope are parameters.

3.3 Statements

Initializers for function definitions are executable blocks, e.g. sequence of statements.

Most statements consist mainly of expressions that are evaluated for their effects. The internal language does not make a distinction between a `case` labeled-statement and an ordinary labeled-statement that may be subject of a `goto`-statement. The label can be any constant expression, not just integral constant expressions as in ISO C++. A block is a sequence of statements with an optional sequence of exception handlers. In particular, there is no distinction between a `try`-block and other blocks as is done in the concrete syntax of ISO C++. As in ISO C++, the "condition" of a `do`, `while`, or

$$\Sigma = \Sigma^{\text{namespace}} \mid \Sigma^{\text{class}} \mid \Sigma^{\text{function}} \mid \Sigma^{\text{template}} \mid \Sigma^{\text{local}} \mid \Sigma^{\text{prototype}}$$

$$\frac{n : (\Gamma, \tau, \_) \quad \tau \in \mathscr{O} \quad \Gamma \in \Sigma^{\text{namespace}} \cup \Sigma^{\text{local}}}{\Gamma \overset{\text{decl}}{\vdash} n \in Var_\tau}$$

$$\frac{n : (\Gamma, \tau, \sigma) \quad \tau \in \mathscr{O} \quad \Gamma \in \Sigma^{\text{class}} \quad \sigma \ni \texttt{static}}{\Gamma \overset{\text{decl}}{\vdash} n \in Var_\tau}$$

$$\frac{n : (\Gamma, \tau, \_) \quad \tau \in \mathscr{O} \cup \mathscr{R} \quad \Gamma \in \Sigma^{\text{class}} \quad \sigma \not\ni \texttt{static}}{\Gamma \overset{\text{decl}}{\vdash} n \in Data_\tau}$$

$$\frac{n : (\Gamma, \tau, \_) \quad \tau \in \mathbb{T} \quad \Gamma \in \Sigma^{\text{template}} \cup \Sigma^{\text{prototype}}}{\Gamma \overset{\text{decl}}{\vdash} n \in Param_\tau}$$

$$\frac{n : (\Gamma, \tau, \_) \quad \tau \in \mathscr{R} \quad \Gamma \in \Sigma}{\Gamma \overset{\text{decl}}{\vdash} n \in Ref_\tau}$$

$$\frac{n : (\Gamma, \tau, \_) \quad \tau \in \mathscr{F}}{\Gamma \overset{\text{decl}}{\vdash} n \in Fun_\tau}$$

$$\frac{n : (\Gamma, \tau, \_) \quad \Gamma \in \Sigma^{\text{class}}}{\Gamma \overset{\text{decl}}{\vdash} n \in Member_\tau}$$

$$\frac{\chi : (\Gamma, \Pi[\![p_1 : \mathbb{T}_1, \ldots, \times p_n : \mathbb{T}_n]\!] \twoheadrightarrow e) \quad e \in \tau}{\Gamma \overset{\text{decl}}{\vdash} \chi \in \mathbb{T}_1 \times \cdots \times \mathbb{T}_n \twoheadrightarrow \tau}$$

Figure 4: Typing judgment rules for declarations

$s ::=$		*statements*
	`break` | `continue`	
	$\mid \mathcal{R}eturn(e)$	*jump statement*
	$\mid \mathcal{G}oto(n)$	
	$\mid \mathcal{E}xprStmt(e)$	*expression-statement*
	$\mid \mathcal{L}abeledStmt(e,s)$	*labeled-statement*
	$\mid \mathcal{B}lock(s_{seq}, h_{seq})$	*compound statement*
	$\mid \mathcal{I}f(e \mid d, s, s)$	*selection statement*
	$\mid \mathcal{S}wicth(e \mid d, s)$	
	$\mid \mathcal{W}hile(e \mid d, s)$	*iteration statement*
	$\mid \mathcal{D}o(e \mid d, s)$	
	$\mid \mathcal{F}or(e \mid d, e \mid d, e, s)$	
	$\mid d$	*block-declaration*
$h ::=$		*catch-block*
	$\mathcal{C}atch(d, s)$	

Figure 5: Statements

`for`-loop can be a declaration, as long as that declaration has an expression value convertible to the ISO C++ boolean type `bool`. Such constructs happen for example in codes like

```
if (ast::call* c = dynamic_cast<ast::call*>(x)) {
   // x is a ast::call, use as c.
}
else {
   // x is not a ast::call.
}
```

And finally, "simple" declarations are considered statements. The operational semantics of statements will be reported elsewhere; for the purpose of this paper — especially for the illustration of abstract typing of templates — it is sufficient to know that selection and iteration statements accept declarations in the "condition" place, as long as they have expression value convertible to `bool`.

3.4 Expressions

The rules for the abstract syntax of C++ expressions is given in Figure 6. C++ expressions are constructed out of literals, declared identifiers, with unary, binary and ternary operators (see Figure 6). For generality and uniformity, we consider type expressions (§3.1) and statements (§3.3) also as expressions.

The requirement that a name used in an expression must have a prior declaration obviously translate to the fact that set $\mathcal{L}(\Gamma, n)$ must not be empty. If the set $\mathcal{L}(\Gamma, n)$ has more than one element then the name n is said overloaded. Only function names can be overloaded. Overload resolution (§4.2) decides on the particular declaration of a name used in a call. Furthermore, the category of C++ expressions with function

$e ::=$		*expressions*
	l	*literals*
	$\mid n$	*names*
	$\mid \mathsf{uop}(e)$	*unary expression*
	$\mid \mathsf{binop}(e,e)$	*binary expressions*
	$\mid \mathsf{terop}(e,e,e)$	*binary expressions*
	$\mid \tau$	*type expressions*
	$\mid s$	*statements*

$\mathsf{unop} : \mathcal{P}ostInc, \mathcal{P}ostDec, \mathcal{P}reInc, \mathcal{P}reDec, \mathcal{D}eref, \mathcal{A}ddress, \mathcal{N}ot, \ldots$

$\mathsf{binop} : \mathcal{P}lus, \mathcal{M}inus, \mathcal{T}imes, \mathcal{D}iv, \mathcal{E}q, \mathcal{N}eq, \mathcal{S}copeRef, \ldots$

$\mathsf{terop} : \mathcal{N}ew, \mathcal{C}onditional$

Figure 6: Abstract syntax of C++ expressions

or object types is in turn divided into two sub-categories: *lvalue* and *rvalue* expressions. For all practical purposes, an lvalue expression is defined as a valid operand of the built-in *address-of* (`&`) operator. An rvalue expression is one that is not an lvalue expression. Many important ISO C++ rules are curiously expressed in terms of that distinction. Consequently, our extended type system include the attribute lvalue to indicate lvalueness.

The type of C++ expressions are controlled by precise language rules [ISO03, Clause 5]. The typing rules are summa-

rized in in Figure 7. A sentence of the form $\Gamma \overset{\text{exp}}{\vdash} e \in E$ means "the expression e has property E in the typing environment Γ." A sentence of the form $\Gamma \overset{\text{exp}}{\vdash} e \nearrow \tau$ means "in the typing environment Γ, the expression e has synthesized type τ." The synthesis is done for symbols by looking up the declaration in the environment.

Literals have types dictated by ISO C++ rules [ISO03, §2.13]. Those type assumptions are part of all initial typing environments. Identifiers need to be visible before use. A variable, a reference, a function parameter name, or a function name evaluates to an lvalue. Function application follows the usual rule: if a call has signature $(\tau_1, \ldots, \tau_n)$ and each of the arguments $(e_1, \ldots, e_n)$ is convertible to expected type, then the call is well formed and has the type specified. Note that the conversions considered in the internal form are those necessary to implement the standard call-by-value and call-by-reference semantics. More elaborated notions of conversion will be considered in §4.1 for the external language.

$$\frac{l \in \mathit{literal}}{\Gamma \overset{\text{exp}}{\vdash} l \nearrow \tau_l} \qquad \frac{\mathcal{L}(\Gamma, x) = \emptyset}{\Gamma \overset{\text{exp}}{\vdash} x \in \bot}$$

$$\frac{\Gamma \overset{\text{decl}}{\vdash} x \in \mathit{Var}_\tau \quad \tau \in \mathscr{O}}{\Gamma \overset{\text{exp}}{\vdash} x \nearrow \tau \cap \mathit{lvalue}} \qquad \frac{\Gamma \overset{\text{decl}}{\vdash} x \in \mathit{Ref}_{\mathit{ref}(\tau)} \quad \tau \in \mathscr{O} \cup \mathscr{F}}{\Gamma \overset{\text{exp}}{\vdash} x \nearrow \tau \cap \mathit{lvalue}}$$

$$\frac{\Gamma \overset{\text{decl}}{\vdash} x \in \mathit{Param}_\tau \quad \Gamma \overset{\text{type}}{\vdash} \tau \in \mathscr{O}}{\Gamma \overset{\text{exp}}{\vdash} x \nearrow \tau \cap \mathit{lvalue}}$$

$$\frac{\Gamma \overset{\text{decl}}{\vdash} \_(x, \tau)}{\Gamma \overset{\text{exp}}{\vdash} x \in \tau}$$

$$\frac{\Gamma \overset{\text{type}}{\vdash} f \nearrow (\tau_1, \ldots, \tau_n) \to \tau \quad \forall i\, e_i \searrow \tau_i}{f(e_1, \ldots, e_n) \in \tau}$$

$$\frac{\Gamma \overset{\text{type}}{\vdash} e \nearrow \tau \cap \mathit{lvalue}}{\Gamma \overset{\text{type}}{\vdash} e \searrow \mathit{ref}(\tau)} \qquad \frac{\Gamma \overset{\text{type}}{\vdash} e \nearrow \tau \cap \mathit{lvalue}}{\Gamma \overset{\text{type}}{\vdash} e \searrow \tau}$$

$$\frac{\Gamma \overset{\text{decl}}{\vdash} x \in \mathit{Var}_\tau \quad x \in \mathit{fv}(e) \quad \Gamma \overset{\text{decl}}{\vdash} y \in \mathit{Ref}_\tau}{\Gamma \vdash e \asymp [y/x]e}$$

$$\frac{\Gamma \overset{\text{decl}}{\vdash} x \in \mathit{Ref}_\tau \quad x \in \mathit{fv}(e) \quad \Gamma \overset{\text{decl}}{\vdash} y \in \mathit{Var}_{\tilde{\tau}}}{\Gamma \vdash e \asymp [y/x]e}$$

Figure 7: Typing rules for expressions

The attentive reader may wonder why we have given only the rules for function application and not for other operators. Conceptually, C++ operators are just functions the implementation of which can come from two sources: built-in or user-defined. So, the typing rules for built-in operators are essentially no different from those of user-defined operators, which are the rules for function application. The types for operators with built-in meanings are specified in [ISO03, §13.6]. Finally, substituting a variable for a free reference in an expression does not change the type or meaning of that expression. And vice versa. That property is just for the implementation of call-by-reference. Concretely, it means that there is no distinction exists between the *use* of a variable of type τ and the use of a reference of type $\mathit{ref}(\tau)$.

4 The external language

As stated previously, C++ is an explicitly typed language in its intrinsic form. However, programming languages that require explicit type annotation on just about every expression are impractical for real-world programming because they put an annotation burden on the programmers that ultimately leads to unreadable, unmanageable programs [PT98]. The external form of C++ makes essential use of program contexts to relieve the programmer from the burden of most annotations.

4.1 Conversions

A fundamental ideal of C++ is to provide equal support for user-defined types and built-in types.

Implicit conversions play an essential rôle in the interpretation of expressions. They happen essentially in assignment operations and initializations (definitions, calls or `return`-statements).

In addition of standard built-in conversions (e.g. integer promotion, floating point promotion, etc.), a programmer can define conversions between user-defined types and between user-defined types and built-in types. The set of standard conversions is precisely defined in [ISO03, §4], the exact details of which are not important for the discussion that will follow. The most relevant information from there are:

- conversions are ranked;
- a *standard conversion sequence* is a finite sequence of standard conversion where the conversions are sequenced in a specific order.

A *user-defined conversion sequence* is a triple $(\mathsf{S}_i, \mathsf{U}, \mathsf{S}_t)$ where both S_i and S_t are standard conversion sequences and U is a user-defined conversion. An implicit conversion is either a standard conversion sequence or a user-defined conversion sequence. The ISO C++ standard has a detailed set of rules [ISO03, Clause 4 and §13.3.3] that describes the computation of implicit conversions. For the purpose of this paper, we will assume a given operation, $\mathsf{Convertible}$, such that for an expression e and a type τ in a typing environement Γ the term $\mathsf{Convertible}_\Gamma(e, \tau)$ yields

1. either $\bot$ (failure), when the expression is not implicitly convertible to τ; or
2. (a) a typing environment $\Gamma' \supseteq \Gamma$ and a rewrite rule $e \rightsquigarrow e'$ such that
$$\Gamma' \overset{\text{exp}}{\vdash} e' \in \tau.$$
   (b) a conversion rank that describes the rewrite from e to e'.

The new environment Γ' may contain new bindings or declarations necessary to make the typing well-formed. This is the case when the implicit conversion involves a user-defined conversion, obtained from implicit instantiation of a converting constructor or conversion function. Where there

is no confusion about the type environments, we just write $e \searrow e' \in \tau$ to say that the expression converts implicitly to e' with type τ; or even more succintly, $e \searrow \tau$ to mean that the expression e is implicitly convertible to τ.

For a given type τ, the set $\searrow \tau$ of expressions implicitly convertible to τ, can be interpreted as a type. That view, which is complementary to conventional type-checking, was developed in [Str03] and carried on in the specification of concepts [DRS05]. In C++ compilers, the concrete manifestation of $\mathsf{Convertible}_\Gamma(e,\tau) = \bot$ is usually in form of diagnostic messages.

4.2 Overload resolution

When name-lookup $\lfloor(\Gamma, n)$ finds more than one declarations specifying different types, the name n is said to be overloaded. ISO C++ allows only overloading for function or operator names. However, our framework can describe overloading for general symbols and values too.

Function overloading is fundamental to programming in C++. In syntactic contexts where an operator is involved or a function is called, the compiler infers a set of matching or viable functions, *i.e.* set of function declarations that have enough parameters and such that each argument is convertible to the corresponding parameter type. Given an expression list $(a_1, \ldots a_n)$, a function declaration

$$f \in (\tau_1, \ldots, \tau_m) \to \tau$$

is a match if

- $n \leq m$, and for $i > n$, the parameter at position i has a default argument;
- for $1 \leq i \leq n$, the expression e_i is implicitly convertible (§4.1) to τ_i.

The ranking of implicit conversions (§4.1) and the partial ordering of function templates (§5.2) induce a partial ordering on matching functions. Given two such functions f and g, we say that f is a better match than g, if:

- the conversion ranks for f are no worse that those of g, then there is at least one argument for which f realises a better conversion. Otherwise,
- f is a non-template function and g is generated from a template; otherwise
- both f and g are generated from templates F and G, and the function template F is more specialized than G (see §5.2).

The set of viable or matching functions is ordered as stated above, and the *best* viable function — if it exists — is chosen. See [ISO03, §13.3.3] for the ISO C++ formulation of the rules.

For the purpose of this article, we'll asume given a primitive operation, named *resolve*, such that: given a symbol f, an argument list $(a_1, \ldots, a_n)$, and a typing environment Γ the term $\mathit{resolve}_\Gamma(f, (a_1, \ldots, a_n))$ yields:

1. either $\bot$ (failure) when $\lfloor(\Gamma, f) = \emptyset$ or no best viable candidate exists; or
2. a collection of rewrite rules $a_i \rightsquigarrow a_i'$, a new typing environment $\Gamma' \supseteq \Gamma$, and a substitution $f \mapsto \delta_f$ prescribing which declaration of the symbol f is to be chosen.

Contrary to the type class mechanism in Haskell, C++ does not require users to declare a global repository where members of an overload set should be manually registered. The members of an overload set need not share a "generic instance" relationship. While the common source of overloading is multiple declarations for a name with different types in the same scope, function declarations may also be overloaded as the result of argument-dependent name lookup whereby the name of a function is also looked in scopes associated with the arguments. Such look-up mechanism defeats a repository-based overloading scheme.

A difference between overloading and overriding, is that the function selected by the overload resolution *is* the actual function that will be executed, not a different function declared after the point of call that may probably have been a better match. From that, it must be clear that overloading relies purely on static type information available at the call site.

4.3 Typing rules

This section extends the core typing judgment rules, displayed in Figure 7, to account for implicit conversions and function overloading.

$$\frac{\mathit{resolve}_\Gamma(f,(e_1,\ldots,e_n)) = \{\delta_f\} \quad \Gamma \overset{\text{exp}}{\vdash} \delta_f \nearrow (\tau_1,\ldots,\tau_n) \to \tau}{\Gamma \overset{\text{exp}}{\vdash} f(e_1,\ldots,e_n) \nearrow \tau}$$

$$\frac{\Gamma \overset{\text{exp}}{\vdash} e \nearrow \tau \cap \mathit{lvalue}}{\Gamma \overset{\text{exp}}{\vdash} e \searrow p \in \mathit{Param}_{\mathit{ref}(\mathit{const}(\tau))}}$$

Figure 8: Typing rules for calls

The first rule is a summary of the overload resolution process. Note that in the formulation of synthesized type, the function need not have the exact number of parameters as there are arguments. It could have more, but in that case, the extraneous parameters are required to have default values. Since those default values do not participate in the overload resolution process, we can safely ignore their existence during the type elaboration. The last rule says that an lvalue expression of type τ can be implicitly bound to a parameter of type $\mathit{ref}(\mathit{const}(\tau))$.

5 Templates

Over the last decade templates have emerged as a key language feature for building efficient, extensible, and modular mainstream systems and libraries. The efficiency and expressive power offered by templates is based on two factors. First, template instantiations combine information available at both definition and instantiation contexts. Second, templates are typically implicitly instantiated. A template is implicitly instantiated if and only if it is used in a way that is essential to the program semantics.

A particularity of the C++ template system, both source of expressivity and weakness, is the coupling between template definitions and their uses. For instance, consider the following function template definition part of the standard library:

```
template<class InIter, class OutIter>
OutIter copy(InIter first, InIter last, OutIter out)
{
  for (; first != last; ++first, ++out)
     *out = *first;
  return out;
}
```

A call `copy(begin, end, dst)` will copy the sequence delimited by the interval [`begin`,`end`) to a sequence starting from `dst`; it is assumed that the destination sequence has enough room to receive the input sequence.

The call leads to successful instantiation only if the deduced type arguments ι and ω for the type parameters `InIter` and `OutIter` have the following properties:

1. instances of ι must be copy-initializable, so that they can be used as function arguments in calls to `copy()`;
2. two such instances must be equality comparable in the sense that the expression `first != second` must be valid and its value convertible to the boolean type `bool`;
3. an expression of type ι must support the pre-incrementation operation; similar assumption for expression of type ω;
4. the expression `*out = *first` must be valid (which implies that every sub-expression must also be valid).

For example, it can be used to copy list elements to a vector

```
list<int> l(768);
// ...
vector<int> v(l.size());
copy(l.begin(), l.end(), v.begin());
```

Note that the function template `copy()` does not concern itself with memory management; it assumes that the destination has enough room. But what if it does not? Instead of writing a new function `copy'()` that tries to deal with memory management, it suffices to make sure that the output iterator does the right thing. That is, the output iterator should make room for the new element it is about to receive. Standard iterator adapters are provided for that:

```
list<int> l(768);
// ...
vector<int> v;
copy(l.begin(), l.end(), back_insert(v));
```

Here, `back_insert(v)` constructs an outpout iterator out of `v` such that each time it receives a datum x, it automatically sends it to `v` with `v.push_back`(x), thus handing off memory management to the vector `v` which knows best how to accomplish such tasks for its internals.

The same function `copy()` can also be used to print out the content of a sequence:

```
list<int> l(768);
// ...
copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
```

The function template `copy()` can be used with any combination of iterators that satisfies the assumptions we list above. This versatility of templates is key to their success in mainstream libraries and systems. However, that versatility also causes problems. We propose a solution, a type system for template arguments, called *concepts* in [DRS05], based on the formalism we report in this paper.

5.1 Template argument deduction

In the previous examples, `copy` is a template. Therefore, `copy` needs values for its template parameters for instantiation purpose. However, we specified none. The external form of C++ has a machinery for deducing template arguments in almost all cases. That is, a C++ compiler is required, in many cases, to deduce values for the template parameters of a function by looking only at the type of the arguments in the call.

The formal process of argument deduction can be framed in terms of constraints solving. Consider a function template declaration

$$\Gamma \overset{\text{exp}}{\vdash} f \in \Pi[\![p_1 : \mathbb{T}_1, \cdots, p_m : \mathbb{T}_m]\!] \rightarrowtriangle (\tau_1, \ldots, \tau_n) \rightarrow \tau$$

where the function parameter types τ_i and return type τ use the template paramaters p_j. Deducing values for p_j when f is presented with a call argument list $(a_1, \ldots, a_n)$ is equivalent to solving in parallel the system of constraints

$$\forall i,\ \tau_i \simeq \alpha_i \text{ where } \Gamma \overset{\text{exp}}{\vdash} a_i \nearrow \alpha_i.$$

The relation $\simeq$ is an *almost* type equality relation. The exact details of that almost type equality are spelled out in [ISO03, §14.8.2]. The constraints can be solved either with unification algorithms or any other constraint solving system, yielding a substitution $\mathcal{S}$ the domain of which consists of paramters p_j.

The deduction process fails

- if there is no unique solution, or
- if not all template parameters have deduced values, or
- if the substituted type $\mathcal{S}[(\tau_1, \ldots, \tau_n) \rightarrow \tau]$ is not well-formed, *i.e.* not a member of $\mathscr{T}$.

The requirement that the constraints are solved in parallel, make parameters play symmetrical rôle like in algorithm $\mathcal{W}'$ of Bruce McAdam [McA98], which does not have the left-to-right bias of the classical algorithm $\mathcal{W}$ of Damas and Milner [DM82]. That also implies that call in the fragment

```
template<class T>
void foo(const T*, const T*);

f("hello", 0);
```

will fail whereas it would succeed in

```
void foo(const char*, const char*);
f("hello", 0);
```

Such failing constructs are mostly regarded as curiosities by C++ programmers. However, the error message a compiler produces varies greatly from the most obscure to the most informative.

Recall that C++ template parameters are not restricted to be type parameters. They can be any compile-time values. For example the following is well known among C++ programmers:

```
template<class T, int N>
int size(const T (&)[N])
{
  return N;
}

template<class T, int N>
T* begin(T (&a)[N])
{
  return &a[0];
}

template<class T, int N>
T* end(T (&a)[N])
{
  begin(a) + size(a);
}
```

The function template `size` can deduce the number of elements in a C-array. It is a far more type safe alternative to C-preprocessor macro-based tricks that are sometimes used. The function templates `begin` and `end` provide functionalities similar to those of the standard containers, so that we can simply write

```
string options[] = { // bound computed by the compiler
  // ...
};

copy(begin(options), end(options),
     ostream_iterator<string>(cout, "\n"));
```

Please note that integer template arguments are not some obscure minor language feature; widely used libraries rely on them and their use is an essential part of what is often called template meta-programming.

5.2 Partial ordering of templates

There are various circumstances (usually in overload resolution) where the need arises to locate from a set of function templates the best match for a given template argument list. This is accomplished through the notion of *partial ordering* of function templates.

Given two template declarations

$$\Gamma \overset{\text{exp}}{\vdash} f \in \Pi[\![p_1 : \mathbb{T}_1, \cdots, p_m : \mathbb{T}_m]\!] \twoheadrightarrow (\tau_1, \ldots, \tau_n) \to \tau$$

$$\Gamma \overset{\text{exp}}{\vdash} f' \in \Pi[\![p'_1 : \mathbb{T}'_1, \cdots, p'_m : \mathbb{T}'_m]\!] \twoheadrightarrow (\tau'_1, \ldots, \tau'_n) \to \tau'$$

the template f is said to be *at least as specialized as* the template f', writtem $f \preccurlyeq f'$, if template argument deduction for f' against the parameter-type list $(\tau_1, \ldots, \tau_n)$ succeeds with type equality, instead of almost type equality (§5.1). Note that the requirement is not that there exist an arbitrary substitution $\mathcal{S}$ that makes $(\tau'_1, \ldots, \tau'_n)$ identical to $(\tau_1, \ldots, \tau_n)$. The requirement is that template argument deduction succeeds, which is a more stringent requirement.

The function template f is said *more specialized* than f' if we have the simultaneous conditions

$$f \preccurlyeq f' \text{ and } f' \not\preccurlyeq f.$$

In which case, we simply write $f \prec f'$.

During overload resolution, if two functions compete with similar conversion ranks, and both come from templates, then the function coming from the most specialized template — when it exists — is prefered. We emphasize that in selecting the most specialized template, it is not just a matter of pattern matching where the first match or the most recently defined specilization is taken. Rather, there is a global partial ordering and an attempt is made to chose the best when it exists. Consider the fragment:

```
template<typenamem T>
void swap(T&, T&);                    // #1

template<typename U>
void swap(vector<U>&, vector<U>&);  // #2

int main() {
  vector<int> v;
  vector<int> w;
  // ...
  swap(v, w);                         // calls #2
}
```

Both declarations of `swap` are part of the C++ standard library. They are overloaded declarations. Both produce instantiations that are viable functions for the call to exchange `v` and `w`, with *exact match*. Partial ordering of templates ranks the second declaration as more specialized than the first declaration. For, no value can be deduced for the template parameter `U` such that the parameter list `(vector<U>&, vector<U>&)` become identical to `(T&, T&)`. On the other hand, it is obvious that `T` = `vector<U>` is a deduction that makes `(T&, T&)` equal to `(vector<U>&, vector<U>&)`.

5.3 Template specializations

Unlike Standard ML [MTHM97], or Haskell [PJ03] C++ allow partial and full specializations of templates. Such language features come from consideration of practical systems. They are used to cope with irregularities of the semantics of language features inherited from C (notably pointers and arrays) and to provide more efficient implementations of general algorithms for specific template argument types (e.g. pointers where general iterators are allowed). Specializations are supplied by users for the system to use when an instantiation is requested; they do not take part in overload resolution. For instance, consider the general class template from [Str00]

```
template<typename T>
class Vector {
public:
// ...
};
```

One can easily see that on a computing environment where all pointer to objects have same representation, every instantiation `Vector<A*>` where `A` is an object type will lead to a separate, different instantiation, thsu duplication of essentially

the same representation. C++ programmers have learnt to factorize such duplicates as

```
template<>                // specialized for void*
class Vector<void*> {
public:
// ...
};

template<typename T>      // then for all other pointers
class Vector<T*> : private Vector<void*> {
public:
// forward functions to Vector<void*>
// ...
};
```

The class `Vector<void*>` is called an explicit of full specialization of the template `Vector`. When an instantiation is requested for `Vector` with template-argument list `<void*>`, this implementation is used instead of generating a class out of the primary template. The second declaration, on the other hand, is still a class template. It is a class template that provide implementations for arguments to `Vector` that are of the form `<T*>` where `T` is a type.

Partial specialization may be conceived of as a form of overriding, but it is *not* overriding as commonly used. Indeed, first, a class template is not a function between objects. It is only a function that maps compile-time entities to declarations. Secondly, the choice of a specializations is based on the set of templates availabe at the point of instantiation, while overriding in the ordinary sense takes into account the object dynamic type and the executed function need not be present at the call site. For instance, the following fragment is invalid according to ISO C++ rules

```
template<class T>
struct X {
  // ...
};

X<int*> x;        // use primary template X, T=int*

template<class T>
struct X<T*> {    // specialize primary X.  This is
                  // invalid, for it would have been
                  // chosen, did it come before X<int*>.
  // ...
};
```

A request of an instantiation of a template χ with template-argument list results in chosing the most specialized template, when it exists. Otherwise, the program is in error.

5.4 Templates and conventional type systems

Several programming languages that support generic programming also offer some forms of separate checking. So, why are the solutions adopted by those languages are not good enough for C++?

C++ template-arguments are not restricted to types. Compile-time constants (e.g. *values*) can be template arguments too. A useful type system for template cannot not just be a variation of kind systems as found in some modern programming languages. Indeed, in such systems, a kind is the type of all types. A kind system define some structures in the space of type constructors. Kind expressions are usually given by the grammar

$$\kappa ::= \natural \mid \kappa \to \kappa.$$

C++ templates are not (just) type generators. They are declaration generators. So, `copy` generates function declarations when given appropriate type arguments. If we were to carry traditional kind systems to C++ templates, then the type of the template `copy` would have kind $\natural \times \natural \to \natural$ which obviously is insufficient to describe expectations on its template-arguments. The reason being that it only says what sorts of arguments the template type expects and what the type of its instantiation is. It does not say anything about the actual assumptions made on the type parameters `InIter` and `OutIter`.

Furthermore, a type system for template parameters cannot just be a dependent type system [Aug98, XP99] as no checking is done at runtime and not every value can be template arguments. Only compile time constants can be template arguments. Moreover, in such dependent type systems `fill` would have type

$$(\texttt{InIter} : \natural, \texttt{OutIter} : \natural,\\ \texttt{first} : \texttt{InIter}, \texttt{last} : \texttt{InIter}, \texttt{out} : \texttt{OutIter}) \to \texttt{OutIter}$$

which still is insufficient to describe the assumptions on the template-arguments. While the dependent type here captures some aspects of the dependencies of the parameters, it also says nothing about the crucial assumptions on `FwdIter` and `T`.

Finally, since expressions inside template definitions involve intricate combination of properties of template arguments, templates challenge most of the advanced dependent type systems currently used as basis for research programming languages.

5.5 Abstract typing of template definitions

The previous formalism developed for conversion and overloading can be deployed to annotate the body of template definitions. Most C++ compilers usually do not annotate expressions in template definitions (except where the language rules require it) because they don't know actual types of function parameters and such, thus defering the annotation to instantiation time.

One can do better than that. Indeed, most of the rules in fig:expr:typing and Figure 8 can be applied to expressions in template definitions, by introducing local type variables. When faced with a call

$$f(e_1, \ldots, e_n)$$

where the argument list contains expressions whose type depende on template parameters, the compiler can synthesized a type for the call:

1. Introduce type variables to type f:

$$\Gamma \overset{\text{exp}}{\vdash} f \nearrow (\tau_1, \ldots, \tau_n) \to \tau \text{ where } \tau_1, \ldots \tau_n, \tau \text{ are fresh,}$$

2. generate the constraints

$$e_i \searrow \tau_i \quad \forall i,$$

3. then synthesize a type for the call

$$\Gamma \overset{\text{exp}}{\vdash} f(e_1, \ldots, e_n) \nearrow \tau.$$

The approach sketched above can be the basis of systematic constraint set generation out of template definition, than can be separately used to implement enforcement of conceptual requirements on template parameters in template definitions.

6 A concept system

The abstract typing exemplified in the preceeding section is taken as the basis of the template systems we propose in [DRS05]. However, it should be noted that concepts are not just set of constraints, as they can used as predicates e.g. types, therefore constitute a fondation for improved overloading of function templates and partial specializations.

7 Related work

There have been several contributions to describe the semantics of the C or C++ programming languages. The work of Ravi Sethi [Set80] appears to be among the the earliest formalizing C. Yuri Gurevich and James Huggins [GH93] proposed a denotational semantics for C; C++ was not considered. Charles Wallace [Wal95] considered pre-ISO Standard C++; his work did not provide a uniform and general description. In contrast to this paper, templates were not recognized as central part of C++. Kathleen Fischer and John Mitchell [FM94, FM95] considered formal models of object-oriented languages features that could account for inheritance and overriding in C++. However, their work did not contain a formalization of the C++ template system and overloading. They suggested a form of F-bounded or higher-order quantification to constrain template arguments. Such constraints do not appear to conveniently cope with contemporary C++ template usage in mainstream libraries as discussed in the concept work [DRS05].

There has been a large body of work in the functional programming community to support overloading. We will briefly mention the contributions of Stephan Kaes [Kae88, Kae92], Philip Wadler and Stephen Blott [WB89], Tobias Nipkow and Christian Prehofer [NP93], Mark Jones [Jon94], Satish Thatté [Tha94], Cordelia Hall and collaborators [HHPJW96], Simon Peyton Jones *et al.* [PJJM97], Peter Stuckey and Martin Sulzman [SS02]. They have mostly focused on extending the classical Hindley-Milner type systems with some notions of constraints. A characteristic of systems with overloading is that the members of the overload set share a relationship of *generic instances* which makes them much closer to the notion of overriding and template specializations than overloading in C++. Manuel Chakravarty and collaborators [CKPJM05, CKPJ05] recently proposed to extend Haskell with associated types, thus moving closer to C++ notion of template specialization and usage in building efficient libraries and systems.

8 Conclusion and future work

We can formally describe ISO C++ in a way that has proven useful for defining an exceptionally simple complete typed abstract syntax tree representation of C++ codes and can be used to precisely reason about existing and proposed language features. This description appears to be new. Contrary to previous works that confined themselves to small subsets of C++, our aim has been a system that is a *superset* of ISO C++, general enough to account for proposed extensions for C++0x. We have also linked to and drawn constrast some approaches to the notion of overloading, e.g. type classes, notably in the functional programming languages like Haskell and extensions to Hindley-Milner type systems. The formal description has also served as basis for the design of a type system for templates. The low-level parts and the C heritage will be subject of future work.

9 References

[Aug98] Lennart Augustsson. Cayenne — a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[Aus98] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, October 1998.

[CKPJ05] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated Type Synonyms. In *ACM SIGPLAN International Conference on Functional Programming*, Tallinn, Estonia, 2005. ACM Press.

[CKPJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated Types with Class. *ACM SIGPLAN Notices*, 40(1):1–13, January 2005.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 207–212, 1982.

[DRS05] Gabriel Dos Reis and Bjarne Stroustrup. Specifying c++ concepts. Technical Report D1886=05-0146, ISO/IEC SC22/JTC1/WG21, July 2005.

[FM94] Kathleen Fischer and John C. Mitchell. Notes on Typed Object-Oriented Programming. In *Proceedings of Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 844–885. Springer-Verlag, 1994.

[FM95] Kathleen Fischer and John C. Mitchell. The Development of Type Systems for Object-Oriented Languages. *Theory and Practice of Object System*, 1(3):189–220, 1995.

[GH93] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language. In *Selected papers from CSL'92 (Computer Science Logic)*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308, 1993.

[HHPJW96] Cordelia V. Hall, Kevin Hammond, Simon Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Lan-*

guages and Systems, 18(2):109–138, 1996.

[ISO98] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 1998.

[ISO03] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.

[Jon94] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[Kae88] Stefan Kaes. Parametric Overloading in Polymorphic Programming Languages. In *Proceedings of the 2nd European Symposium on Programming*, volume 300 of *Lecture Notes In Computer Science;*, pages 131–144. Springer-Verlag, 1988.

[Kae92] Stephan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 193–204, 1992.

[LK00] Angelika Langer and Klaus Kreft. *Standard C++ IOStreams and Locales*. Addison-Wesley, January 2000.

[McA98] Bruce J. McAdam. On the Unification of Substitutions in Type Inference. In *Proceedings of the 10th International Workshop, IFL'98*, volume 1595 of *Lecture Notes in Computer Science*, pages 137–152, 1998.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[NP93] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 409–418, Charleston, South Carolina, United States, 1993.

[PJ03] Simon Peyton Jones. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[PJJM97] Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: exploring the design space. In *Proceedings of the Haskell Workshop*, Amsterdam, The Netherlands, June 1997.

[PT98] Benjamin C. Pierce and David N. Turner. Local Type Inference. In *Symposium on Principles of Programming Languages*, pages 252–265, San Diego CA, USA, 1998. ACM.

[Set80] Ravi Sethi. A case study in specifying the semantics of a programming language. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 117–130, Las Vegas, Nevada, 1980.

[SL94] Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report N0482=94-0095, ISO/IEC SC22/JTC1/WG21, May 1994.

[SS02] Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM SIGPLAN Notices*, 37(9):167–178, September 2002.

[Str] Bjarne Stroustrup. C++ Applications. `http://www.research.att.com/~bs/applications.html`.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[Str03] Bjarne Stroustrup. Concept checking — A more abstract complement to type checking. Technical Report N1510, ISO/IEC SC22/JTC1/WG21, September 2003.

[Tha94] Satish R. Thatté. Semantics of type classes revisited. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 208–219, Orlando, Florida, United States, 1994. ACM Press.

[Wal95] Charles Wallace. *Specification and validation methods*, chapter The semantics of the C++ programming language. Oxford University Press, Inc., 1995.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, Austin, Texas, USA, 1989.

[XP99] Howgwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.