TOOL SUPPORT FOR AXIOMATIC PROGRAMMING

A Senior Scholars Thesis

by

CARLA VILLORIA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the designation of

UNDERGRADUATE RESEARCH SCHOLAR

April 2010

Major: Computer Science

TOOL SUPPORT FOR AXIOMATIC PROGRAMMING

A Senior Scholars Thesis

by

CARLA VILLORIA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the designation of

UNDERGRADUATE RESEARCH SCHOLAR

Approved by:

Research Advisor:	Gabriel Dos Reis
Associate Dean for Undergraduate Research:	Robert C. Webb

April 2010

Major: Computer Science

ABSTRACT

Tool Support for Axiomatic Programming. (April 2010)

Carla Villoria
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Gabriel Dos Reis
Department of Computer Science and Engineering

Many problems arising from spectacular error messages involving C++ templates are related to the fact that assumptions made by the C++ standard algorithms are stated in informal comments, not in code that is checked by the compiler. Some of those properties are syntactic, meaning that the compiler can do syntax and type checking and reject erroneous constructs. Others are semantics, e.g. that a type is regular. Such assumptions can be checked only if programmers have ways to express those assumptions in code.

My project proposes the use of two features in C++, *concepts* and *axioms*, that would allow programmers to express semantic requirements in code. To show the benefits that this approach could have we have developed an interpreter, Liz, capable of handling a subset of C++ augmented with *concepts* and *axioms*.

Liz has been successful in demonstrating how these new features could save a lot of time and effort to programmers, and more importantly, how they could make *templates* less intimidating, more accessible, and truly mathematically accurate.

DEDICATION

To my parents, who have instilled in me a passion for knowledge and discovery.

ACKNOWLEDGMENTS

I would primarily like to thank my research advisor, Dr. Gabriel Dos Reis, as without him this project would not have been possible. Thanks to his constant support and advice I was able to keep going even in the hardest times.

I would also like to thank Paul McJones and Sean Parent for their suggestions and feedback throughout the process of writing this thesis.

Finally, a big thank you goes to my family and friends, whose encouragement and love kept me focused on my goals.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Over the last decade, thanks to the trail blazing work of Alex Stepanov (Stepanov & Lee, 1994; Dehnert & Stepanov, 1998; Musser & Stepanov, 1994), the ISO C++ programming language has significantly contributed to making generic programming a sound and viable programming methodology. Generic programming (Musser & Stepanov, 1988; Gibbons & Jeuring, 2003) focuses on useful, practical and efficient abstract procedures, and on how to provide abstractions in order to use a single algorithm in several different implementations. Reusability is a fundamental aspect of this discipline. The C++ programming language directly supports generic programming through templates, special functions that operate with generic types and are reused with different instantiations.

Generic programming with C++ templates

C++ templates are the basis of many programming techniques and a key element in the construction of libraries (Stepanov & Lee, 1994; Siek & Lumsdaine, 1998; Gärtner & Veltkamp, 2007; Veldhuizen, 1998; Czarnecki *et al.*, 2000). The success of templates is mainly due to their flexibility. They allow functions and classes to operate with generic types. They work for built-in types as well as user-defined types without requiring inheritance from some predefined class; they allow independent libraries to be composed. Moreover, templates provide near optimal efficiency, which means, the elimination of function calls in favor of inlining, the avoidance of code generation for unused functions, etc. This

---

This thesis follows the style of Journal of Functional Programming (JFP).

near optimal efficiency is realized by combining information available at both the template definition site and template use sites.

The practice of generic programming, according to Stepanov and Musser (Stepanov & Lee, 1994; Musser & Stepanov, 1994), essentially relies on the notion of *gradual lifting of algorithms*. This process starts with concrete algorithms, and iteratively abstract over various computational aspects until sufficiently general, yet efficient, abstract procedures are obtained. Let's take a look at the following example. Consider you have an array of integers and two pointers, "first" and "last", pointing to the beginning and end of a subsequence of the array, and you want to know the number of elements between these two pointers, you could write the following function:

```
int array_distance(const int* first, const int* last) {
  int n =0;
  while (first != last) {
    first = first+1;
    n = n+1;
  }
  return n;
}
```

On the other hand, if you would like the same functionality for a linked list of integers then you would probably write:

```
int list_distance(const LinkedListNode* first, const LinkedListNode last) {
  int n =0;
  while (first != last) {
    first = first->next;
    n = n+1;
  }
  return n;
}
```

Note that these two programs look very much alike. They differ only in the way one moves from one array slot (or linked list node) to the next. We can avoid writing twice almost the same piece of code by abstracting over some details, e.g. introducing a function parameter

that encapsultes the knowledge of moving from one cell to the next. C++ templates allows a simple expression of this idea:

```
template<typename Iter, typename NextItem>
int distance(Iter first, Iter last, NextItem next_cell) {
  int n = 0;
  while(first != last) {
    first = next_cell(first);
    n = n + 1;
  }
  return n;
}
```

We can now instantiate this abstract procedure with several kind of arguments. For arrays, we need to supply a value for `NextItem` that essentially implements pointer increment. This is done as follows:

```
struct NextIntPointer {
  const int* operator()(const int* p) const {
    return p + 1;
  }
};

int main() {
   int array[] = { 4, 2, 95 };
   return distance(&array[0], &array[2], NextIntPointer());
}
```

This is just as efficient as the original hard-coded function `array_distance`. Similarly, we can recover the function `list_distance` when we use an objet of the class:

```
struct NextLinkedListNode {
  const LinkedListNode* operator()(const LinkedListNode* p) const {
    return p->next;
  }
};
```

In summary, the same abstract procedure `distance` can be instantiated (with appropriate arguments) to different, useful and efficient algorithms, with no loss in efficiency. As you can see, this template example is simple to follow and simple to write.

Flexibility: A two-sided sword

The flexibility offered by templates, although extremely useful as we saw in the previous section, also poses some challenges. As of right now, template definitions are not checked independently of their uses. This implies that most of the type-checking is delayed until template instantiation time. Because of this, successful compilation and linking only shows that the template instantiations were type correct for the arguments used during testing. However, instantiation may fail for other set of arguments that were not tested. Spectacularly poor messages will be result from this delayed checking. The issue is compounded by the fact that the assumptions made by the C++ standard algorithms are stated in informal comments, not in code that is checked by the compiler. In some cases, this can hinder the adoption of Generic Programming methodologies, mainly because these error messages are intimidating to many users. For example, trying to sort a list with the general standard `sort` function

```
list<int> l;
...
sort(l.begin(), l.end());
```

leads to quite obscure and hard to understand error messages (at least with GCC-4.x.y.) In summary, there is currently no simple, convenient, and scalable way to express assumptions that a parameterized algorithm makes on its template parameters.

Axiomatic programming

For generic programming to become mainstream, it is essential that programming languages offer adequate and direct linguistic support. In recent past, there had been promising work, especially in the C++ community, for what is now known as *concepts* (Gregor

et al., 2006; Stroustrup & Dos Reis, 2003a; Dos Reis & Stroustrup, 2006; Stroustrup & Dos Reis, 2003b). However, despite all best efforts, concepts will not be part of C++0x (Stroustrup, 2009b; Stroustrup, 2009a). Among the various novelties that were proposed is the notion of axioms.

An *axiom* (Dos Reis *et al.*, 2009) states what an algorithm implementation may assume of values, but also what properties a user could assume about values. The axiom feature allows us to state in code what we currently state in comments. Concretely, they would permit the addition of requirements on template definitions, and by doing so they would allow the compiler to check (where possible) whether a generic function (or generic class) is used according to the expectations it has on its arguments.

Let's take the `distance` function template. When instantiated with the right combination of arguments, we get useful algorithms. However, mistakes are common, and the following program fragment:

```
std::list<int> l;
distance(array, array + 3, Nextlinkedlistnode());
```

will generate confusing error messages. The reason is that in

```
return p->next;
```

an `int` object is not a structure and does not have `next` member. The real problem here is that we do not have a separate way of saying how the functional parameter `next_cell` relates to the iterator parameters. Axioms and concepts would allow programmers to express those requirements explicitly in the code itself. More specifically, the `distance` function could be written (following page 19):

```
template<typename F>
  requires(Transformation(F))
DistanceType(F) distance(Domain(F) x, Domain(F) y, F f)
{
  //Precondition: y is reachable from x under f
  typedef DistanceType(F) N;
  N n(0);
  while (x != y) {
    x = f(x);
    n = n + N(1);
  }
  return n;
}
```

Take a closer look to the following line:

```
   requires(Transformation(F))
```

This line states that `F` is a transformation on some space. Furthermore, the declaration of the parameters `x` and `y` explicitly says that they are suitable type (as arguments) to the functional parameter `f`.

The interpreter

Alexander Stepanov and Paul McJones recently published a magnificent book (Stepanov & McJones, 2009) on structured generic programming titled *Elements of Programming*. They show-case programming as a mathematical activity, a wonderful journey in the land of simplicity and generality. Their approach makes essential use of *axioms* (and more generally *properties*) and *concepts*. In spite of this reliance on concepts, all of their codes is compilable as almost C++03 (ISO, 2003) program fragments. That is accomplished essentially by use of a few simple macros. In particular, the `requires` "keyword" is actually a C99 variadic macro defined Appendix B.2 to ignore its arguments. Consequently, one of the

benefits of concepts — turning informal descriptions into code so that they can be verified and used for type checking template definitions and uses — is not realized.

In this thesis, I propose to investigate how an interactive tool, in the form of an interpreter for a subset of C++ appropriately extended, supports effective practice of structured programming with axioms as advocated by Alex Stepanov and McJones. This interpreter is called Liz.

CHAPTER II

LIZ

Liz is made of several different components. Each of its parts are responsible for specific tasks; together, they form an interpreter. In this chapter, I will present the internal workings of Liz, which offers linguistic support for the style of structured generic programming advocated in the *Elements of Programming*.

Liz internal design is shown in Fig. 1.

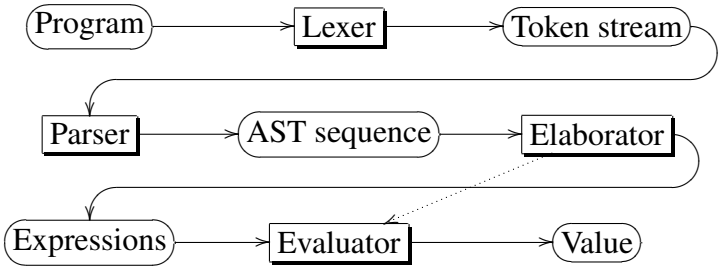

Fig. 1. General Overview of Liz's Architecture

Let me explain what the major components in Fig. 1 mean.

Lexer

Lexing is the process of converting a sequence of characters into a sequence of tokens. The lexer will receive a source program from the user, either directly from standard input or imported as a file. After this, and by following certain rules or regular expressions, the source program will be decomposed into a series of tokens that will eventually become a token stream and will be passed to the parser. In the following function,

```
int id(int x) { return x; }
```

the resulting token stream will be:

"int" id "(" "int" x ")" "" "return" x ";"

where words that are not in quotes represent identifiers, and will be handle different from everything else in the Parser and in subsequent steps throughout Liz. This token stream will be then pass to the parser.

Parser

This section is partially based on a previous report on the Liz parser (Villoria & Dos Reis, 2009). We refer to the reader to that technical report for details that are omitted here. The parser will received the token stream from the Lexer and will convert it into an Ast Sequence following Liz grammar. It is a recursive descent parser, written in Standard C++, using parser combinator technology.

Parser combinators

The grammar and semantics sketch for the subset of C++ used in *Elements of Programming* is described on less than ten pages pp 233–241. This terse description relies on knowledge available elsewhere in the literature. The grammar was designed to be *almost* context-free — this is to be contrasted with Standard C++ grammar which necessitates semantics processing. As explained in p. 239, the only exception is the usual case where a template specialization is explicitly named: an identifier followed by the less-than symbol followed

by an additive expression is a valid production (a relational expression). The ambiguity is resolved by checking whether the identifier names a template. This is the only case where context matters during parsing.

The grammar put forward in the *Elements of Programming* uses the extended Backus–Naur form advocated by Niklaus Wirth (Wirth, 1977). The essential feature of EBNF is the introduction of an explicit iteration construct $\{\mathsf{a}\}$ to stand for $\epsilon|\mathsf{a}|\mathsf{aa}|\mathsf{aaa}|\ldots$ EBNF also introduced optionality of a rule a by $[\mathsf{a}]$. We follow suit. We also introduce shortcuts for some syntactic patterns that appear over and over again. All of these shortcuts are expressible in the EBNF at the expense of repetition and poor abstraction of commonalities. We will use usual alphanumeric identifiers in functional notation for those meta constructions. To keep the notation uniform, we also introduce a functional notation for the essential meta constructs of EBNF:

- ZeroOrMore(a) is EBNF's iteration $\{\mathsf{a}\}$ of rule a.
- OneOf($[\mathsf{a}_1, \mathsf{a}_2]$) is EBNF's choice $\mathsf{a}_1|\mathsf{a}_2$. More generally OneOf($[\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n]$) is $\mathsf{a}_1|\mathsf{a}_2|\ldots|\mathsf{a}_n$ in EBNF notation.

There are places in the grammar where a production is enclosed in "brackets"; for example:

- function argument lists are enclosed in parenthesis
- compound statements are lists of statement enclosed in curly braces
- indexing into an array is an expression in square brackets
- template argument lists are expressions enclosed in angle brackets

All of these instances of bracketing are captured by the combinator Enclosed which can be instantiated with two arguments: the first is a `rule` and the second is a `bracket` token. The instantiation is a rule consisting of the terminal `bracket` followed by `rule`, followed by a matching `bracket`:

```
Enclosed(rule, bracket) =
    bracket rule Closer(bracket)

Parenthesized(rule) =
    Enclosed(rule, OPEN_PAREN)

Bracketed(rule) =
    Enclosed(rule, OPEN_BRACKET)

Braced(rule) =
    Enclosed(rule, OPEN_BRACE)

Angled(rule) =
    Enclosed(rule, LT)
```

The meta combinator Closer is defined by cases on "bracket" tokens:

```
Closer(OPEN_PAREN) = CLOSE_PAREN
Closer(OPEN_BRACKET) = CLOSE_BRACKET
Closer(OPEN_BRACE) = CLOSE_BRACE
Closer(LT) = GT
```

Another combinator that is useful to describe the syntactic structure of expressions is LeftAssociative:

```
LeftAssociative(rule, ops) =
    rule ZeroOrMore(OneOf(ops) rule)
```

This combinator describes binary expressions where the operator associates to the left, as is traditionally the case for additive expressions. A dual combinator, RightAssociative is also defined. Finally, we also used comma-separated items:

```
CommaSeparatedList(rule) =
    rule ZeroOrMore(COMMA rule)
```

Liz's grammar

Toplevel statements

```
toplevel =
    OneOf(template, concept, axiom, structure, procedure, statement)
```

The language described in Appendix B.1 of *Elements of Programming* does not include any production for the toplevel. In particular, no description is given for the structure of a program. This is probably because the book concentrates only on algorithms, e.g. program fragments as opposed to complete applications. However, after the initial completion of this work, Sean Parent indicated that the toplevel was envisioned to consist of *template*, *structure* and *procedure*. We included *statement* at the toplevel because Liz is primarily designed for interactive use and we did not find it a good design to invent an entirely new, different language for interactive uses. We also note that the inclusion of statement at toplevel (which includes simple statements) provides a convenient way to work around the C preprocessor `#include` directive; e.g. we write

```
import("eop.h");
```

instead of

```
#include "eop.h"
```

We are loath to implement a C preprocessor, and we would prefer a compile-time evaluation mechanism for the programming style advocated in *Elements of Programming*.

The grammar for *axiom* and *concept* definitions are not in the *Elements of Programming*. They are our proposal for Liz, and the focus of this thesis. The *concept* production differs from past proposals (Dos Reis & Stroustrup, 2006; Gregor *et al.*, 2006). The rule for *axiom* also differs from what was described in the technical report *Axioms:Semantics Aspects of C++ Concepts* (Dos Reis *et al.*, 2009). It particular, it contains explicit support for universal quantification, instead of relying on indirect encoding through Skolemization.

In this section, we will focus on the grammar of templates, concepts, axioms, structures, and procedures; as some are of the outermost importance, others are new and unexplored, and the rest differ from what is on the *Elements of Programming* in some way or another.

Templates

template =
    TEMPLATE Angled(Optional*(parameter_list)*)
        Optional(*constraint*) OneOf(*axiom, structure, procedure, specialization*)

constraint =
    REQUIRES *condition*

condition =
    Parenthesized(*expression*)

specialization =
    STRUCT *structure_name* Angled(*additive_list*)
        Optional(*structure_body*) **SEMICOLON**

Note that the grammar for template declarations has changed from the *Elements of Programming*. We accept axiom templates. I will show examples of this new feature under the

axioms section.

Concepts

concept =
CONCEPT *identifier* Parenthesized(*parameter_list*)
Braced(ZeroOrMore*(concept_clause)*)

concept_clause =
OneOf(*formula, axiom, procedure*)

Concepts are part of our new proposal. Let take a look at a few examples. The `Transformation` concept p. 17 will be written in Liz as following:

```
concept Transformation(Operation F) {
    UnaryFunction(F);
    Integer DistanceType(Transformation);
}
```

where p. 12,

```
concept UnaryFunction(Function F) {
    Arity(F) == 1;
}
```

Here is another example, the definition of the `HomogeneousFunction` concept p. 12, which is written in Liz as

```
concept HomogeneousFunction(Function F) {
   Arity(F) > 0;
   forall(int i, int j) i < Arity(F) and j < Arity(F) =>
       InputType(F, i) == InputType(F, j);
}
```

Note that in the *Elements of Programming*, there is one more requirement, *Domain*,

```
Regular Domain(HomogeneousFunction T) {
    return InputType(T, 0);
}
```

that we do not take into account in Liz, mainly because we are trying to embodied the *minimal* requirements, and we consider the notion of *homogeneous functional procedure* still properly conveyed by the other two requirements. The function `Domain` is more of a convenience function than an assumption that cannot be derived from the other two assumptions. As ever, there is a balance to strike between minimality and practicality, or convenience.

A more interesting example, `MultiplicativeGroup` p. 68, would be written in Liz as,

```
concept MultiplicateGroup(MultiplicateMonoid T) {
    T multiplicative_inverse(T);
    inverse_operation(multiplicative_inverse,1,*);
}
```

where the meaning of the `inverse_operation` axiom will be discussed further down.

Note that for this example, the *Elements of Programming* also has one more requirement, `operator/`,

```
template<MultiplicativeMonoid T>
    T operator/(T a, T b) {
        return a * multiplicative_inverse(b);
    }
```

that we do not take into account for the same reasons as in the previous example, concept `HomogeneousFunction`.

Axioms

axiom =
AXIOM *identifier* Optional(Parenthesized*(parameter_list)*) Braced(*formula*)

formula =
Braced(*quantifier* Parentized*(parameter_list)*)
OneOf(*proposition,* Braced*(*ZeroOrMore*(concept_clause))*)

proposition =
RighAssociative(*expression, [***IMPLIES***]*) **SEMICOLON**

The axiom constructs lets programmers express properties in a program. As I stated under the *templates* section, in Liz we can define template axioms, or parameterized axioms, and we consider most of the *properties* showed in the *Elements of Programming* representing that — parameterized axioms. We find them useful, not only because they allow us to express simply and elegantly many axioms, but also because they allow us to translate and show general ideas — properties — into code, and make them accessible and easy to use by ordinary programmers. Let me explain this further with examples. The regular_unary_function p.14 property is expressed in the *Elements of Programming* as,

property $(\mathsf{F} : \mathit{UnaryFunction})$

regular_unary_function : F

$$\mathsf{f} \mapsto (\forall \mathsf{f}' \in \mathsf{F})\,(\forall \mathsf{x}, \mathsf{x}' \in \mathrm{Domain}\,(\mathsf{F}))$$
$$(\mathsf{f} = \mathsf{f}' \wedge \mathsf{x} = \mathsf{x}') \Rightarrow (\mathsf{f}\,(\mathsf{x}) = \mathsf{f}'\,(\mathsf{x}'))$$

It would be tempting to write the above property in code as,

```
axiom regular_unary_function(UnaryFunction F) {
     forall(F f1, F f2) forall(Domain(F) x1, Domain(F) x2)
         f1 == f2 and x1 == x2 => f1(x1) == f2(x2);
 }
```

However, this particular expression of the property may not be quite accurate. Let me explain this further. If we had a function twice,

```
int twice(int x) {
    return x + x;
}
```

and we wanted it to follow the axiom `regular_unary_function`, we would call the axiom as,

```
regular_unary_function(twice);
```

This call will create a type checking problem with the parameter type, and the type of the argument `twice`. Furthermore, and more importantly, that definition of the property is saying that all values of type `F` have some property. That is not correct. The concept `UnaryFunction` takes a `FunctionalProcedure`, and this is defined on types, not functions. In concrete terms, the call above would be a type violation because `twice` is a function, not a type. One solution to this problem could be to define the axiom as,

```
axiom regular_unary_function(UnaryFunction F, F f1) {
   forall(F f2) forall(Domain(F) x1, Domain(F) x2)
       f1 == f2 and x1 == x2 => f1(x1) == f2(x2);
 }
```

where we would call,

```
regular_unary_function((int)->int, twice);
```

and it would be type correct, but its readability and scalability would suffer. As we are writing real software for real programmers, we want these two characteristics to be of the outmost importance. In the case above, the first parameter of the call would represent nothing else but a distraction. With that in mind, we conclude that solution would not

be scalable in mainstream use. We would like the system to, in an automated way, take care of deducing the value `(int)->int` for the first axiom parameter, but how? That is when parameterized axioms, also called axiom templates, come into place. Following the exact mathematical expression of the `regular_unary_function` property in the book the translation into Liz would be,

```
template<UnaryFunction F>
    axiom regular_unary_function(F f1) {
        forall(F f2) forall(Domain(F) x1, Domain(F) x2)
            f1 == f2 and x1 == x2 => f1(x1) == f2(x2);
    }
```

However, when we try this piece of code in Liz we run into a couple of issues. One of those is the following error,

```
no match for operation 'Domain' with argument type list (UnaryFunction)
candidate is: Domain:(HomogeneousFunction) -> Regular
```

which occurs during type checking of the uses of type function `Domain` in the definition of the `regular_unary_function` axiom. Notice that earlier, we defined the type function `Domain` as specified in the book,

```
Regular Domain(HomogeneousFunction T) {
    return InputType(T, 0);
}
```

but for the axiom to make sense using the above definition of `Domain` there is an implication in the *Elements of Programming* p. 13 that needs to be followed,

```
(forall FunctionalProcedure F) UnaryFunction(F) => HomogeneousFunction(F)
```

However, it is unclear how the expression `Domain(F)` is well-formed when `F` satisfies only `UnaryFunction`, which is defined unrelated to `HomogeneousFunction`. After discussing this bug with the authors of the book, we decided that the correct solution to this problem was to overload the type function `Domain` for all Unary Functions instead of only for Homogeneous Functions as follow,

```
Regular Domain(UnaryFunction T) {
    return InputType(T,0);
}
```

Moreover, there is still one more issue with this axiom. The English interpretation of the property `regular_unary_function` can be thought of as: "A function applied to equal arguments yields the same result". But if we look at the body of the axiom carefully,

```
forall(F f2) forall(Domain(F) x1, Domain(F) x2)
    f1 == f2 and x1 == x2 => f1(x1) == f2(x2);
```

we can clearly see this is not what is being described. What the above says is: "If two functions, `f1` and `f2`, are equal, and two arguments, `x1` and `x2`, are equal, the result of applying one of those arguments, `x1`, to one function, `f1`, should be equal to applying the other argument, `x2`, to the second function, `f2`." That is very different from our English interpretation of the `regular_unary_function`. If we want to represent, without ambiguities, exactly what the general interpretation is saying we should write the following,

```
template<UnaryFunction F>
    axiom regular_unary_function(F f) {
        forall(Domain(F) x1, Domain(F) x2)
            x1 == x2 => f(x1) == f(x2);
    }
```

which says exactly what we need; and now we could call,

```
regular_unary_function(twice);
```

without it being a type violation and without running into problems along the way.

Now that we have resolved the issues with the `regular_unary_function` axiom we can take a look at the example mentioned in section *Concepts*, `inverse_operation`, that will be written in Liz as,

```
template<Transformation F, Regular T, BinaryOperation Op>
        requires (Domain(F) == T and T == Domain(Op))
    axiom inverse_operation(F inv, T e, Op op) {
        forall(T a) (op(a, inv(a)) == op(inv(a), a) and op(inv(a), a) == e);
    }
```

where the `BinaryOperation` concept will be,

```
concept BinaryOperation(Operation Op) {
    Arity(Op) == 2;
}
```

and the `Operation` concept will be written as,

```
concept Operation(HomogenousFunction Op) {
    Codomain(Op) == Domain(Op);
}
```

Another example of parameterized axioms, identity_element p. 65 would be expressed as,

```
template<Regular T, BinaryOperation Op>
       requires (T == Domain(Op))
   axiom identity_element(T e, Op op) {
      forall(T a) op(a, e) == a and op(e, a) == a;
   }
```

Lastly, the associative p. 31 property will be written in Liz as,

```
template<BinaryOperation Op>
    axiom associative(Op op) {
    forall(Domain(Op) a, Domain(Op) b, Domain(Op) c)
        op(op(a,b),c) == op(a,op(b,c));
}
```

Many people tend to underestimate the value of template argument deduction, but this is of incredible importance for the use and convenience that brings to code organization, composition, and scalability.

Structures

```
structure =
    STRUCT structure_name Optional(structure_body) SEMICOLON

structure_name =
    identifier

structure_body =
    Braced(ZeroOrMore(member))

member =
    OneOf(data_member, constructor, destructor, assign, apply, index, typedef)

data_member =
    expression identifier
        Bracketed(OPEN_BRACKET expression CLOSE_BRACKET)
            SEMICOLON

constructor =
    structure_name Parenthesized(Optional(parameter_list))
        Optional(COLON CommaSeparatedList(initializer)) body

destructor =
    TILDA structure_name Parenthesized() body

construct =
    OPERATOR Braced() Parenthesized(Optional(parameter_list))
        Optional(COLON initializer_list) body

assign =
    VOID OPERATOR EQ Parenthesized(parameter) body

apply =
    expression OPERATOR Parenthesized()
        Parenthesized(Optional(parameter_list)) body
```

index =
    expression **OPERATOR** Bracketed()
        Parenthesized(*parameter_list*) *body*

initializer =
    identifier Parenthesized(Optional*(expression_list)*)

Procedures

procedure =
    expression procedure_name Parenthesized(Optional*(parameter_list)*)
        OneOf(*body,* **SEMICOLON**)

procedure_name =
    OneOf(*identifier, operator*)

operator =
    OPERATOR OneOf(*[***DOUBLE_EQ**, **LT**, **PLUS**, **MINUS**, **STAR**, **SLASH**, **PERCENT***]*)

parameter_list =
    CommaSeparatedList(*parameter*)

parameter =
    expression Optional(*identifier*)

body =
    Compound

Note that in *Elements of Programming*, the type `void` was not considered an expression. However, we believe that a uniform treatment of type expressions benefits from viewing `void` just as any other type expression.

Elaborator

Elaboration is the process of type checking an input source program and de-sugaring it into a simpler language. The Elaborator essentially receives a piece of code that manipulates

in great detail, and at the end, it produces the main matter for the next component in the process, the evaluator. The elaborator brings together the process of type checking a program fragment, and the notion of making explicit what was once implicit in a source code. The specific functions include the generation of code, a medium level language that will eventually get evaluated, and the type checking of a given ast sequence. Sometimes, it can also serve the function of evaluator, but only on type expressions.

Function elaboration

The elaboration of functions is straightforward. We type check the type of the function along with the return type, and then we type check the parameters used in the body of the function. After that, the medium level language code is generated and passed to the evaluator.

Template elaboration

Templates, although similar to functions in many aspects, are handle very differently from them, and their elaboration is far more complicated. Given the fact that the type, or types, used on a template could be diverse, the type checking is not as straight forward, and other things have to be taken into account. For example, take a look at the next two templates:

```
template<typename T, typename U>
    void bar(T,U);


template<typename U, typename T>
    void bar(U,T);
```

According to C++ semantics, these two templates declare the same function. The name of the parameters does not matter much; only their position and consistent uses. The Elaborator has to take into account that the template parameter declaration is far more important than the name they are given. The elaborator looks not only at the names of the typenames, but also at their position in the template-parameter.

Scopes

For the first time in Liz, the notion of scope and scope management come into the picture with the elaborator. This notion is crucial for both the elaborator and the evaluator. A scope can be defined as a map from symbols to values, in a given context; while scope rules tell us how to relate symbols to entities they designate. We use two different types of scopes in Liz, lexical and dynamic. In a lexical scope, the scope of an identifier is fixed at compile time, and if the symbol is not found in a scope, the search continues in its enclosing lexical scope and so on, until either the global scope or a dynamic scope contour is reached. On the other hand, a dynamic scope is implied by a call to a function during run-time, and if a symbol is not found in a given dynamic scope, it is not search in its enclosing dynamic scope; rather, the symbol is searched in the global scope, and if not found, then it is reported as “undefined” or “unbound”.

In the evaluator, we use both lexical and dynamic scope, while in the elaborator we mostly use lexical scope. However, a special case arises when the elaborator deals with type functions and it finds itself in need of evaluating the expression and using an indirect type of dynamic scope. Let me explain this further with the following example,

```
int i = 5;
pointer(int) p = addressof(i);
```

When the elaborator sees `pointer(int)`, it needs to determine what type it designates, and that implies some form of evaluation. It elaborates the expression to determine if it designates a type. Then, it checks whether the full declaration is valid, but for this it needs the value of that expression. That is when it calls the evaluator, which will *implicity* establish a dynamic scope for the purpose of the call evaluator. If the value returned has type `typename`, then we know it is a type, and that the declaration makes sense This is the only time where evaluation, and dynamic scope, take place in the elaborator. At any other time, a lexical scope is used. More concretely, this means that when you see an identifier in the code, you can tell what variable it refers to just by looking at the source code. The meaning of the name of the identifier is determined by which variable binding constructs its used inside. Take a look at the following template,

```
template<typename T>
    T id(T x) {
        return x;
    }
```

when this template gets elaborated there will be four different lexical scopes in place. The first scope, or the global scope, will hold global variables. The second scope will hold the typename and the identifier of the template. The third scope will hold the parameters of the template, and finally, the fourth scope will hold the body.

C++ compatibility

We wanted *Liz* programs to be compatible with C++, but at the same time the aim was to showcase the new features of axioms and concepts, which are not part of C++. While working on the elaborator we realized that in order to add these new features we would have to somehow depart from C++ full compatibility. Take a look at these two templates:

```
template<typename T>
    T id(T x) {
         return x;
    }


template<typename T>
    T twice(T x) {
        return x + x;
    }
```

These two function templates are perfectly valid C++, but let us analyze them. For the function template `id`, the operation applied to parameter `x` is *copy*, and as long as we assume that everything is regular, then it is correct, and it will work with any instantiated type. For the function template `twice` it is different. We are using the operator plus, +, which is defined for some types, but not all. This means that if in C++ we were to call this template with any datatype for which there is no appropriate `operator+` we would get into trouble. This is an important issue in my thesis, and partly explained in the introduction. In Liz, you could write the first program without problems, but the second one will not typecheck. More specifically:

```
no match for call to ‘operator+’ with argument type list (T, T)
candidates are
   operator+: (int, int) -> int
   operator+: (double, double) -> double
```

This error informs the programmer what the problem is, essentially that there is a type checking issue with the expression `x + x`, which implicitly calls the function `operator+`. However, the types of the arguments do not match any known `operator+` in scope, as reported in the "candidates are:" part of the diagnostic. The error also clearly states the reasons for this problem, by specifying that it cannot chose from the list, in this case composed of *int* or *double*, to make the expression type correct. More importantly, Liz does not *trick* the programmer into believing that the template will work with any type template argument. That is when concepts come into action. In Liz, if you write:

```
concept Addable(typename T) {
    T operator+(T,T);
}
```

and then use it to define the function template `twice` as,

```
template<Addable T>
    T twice(T x) {
        return x + x;
    }
```

it will work just fine, and most importantly, it will not generate spectacular error messages if you tried to call the template with a non-correct parameter type.

Types and concepts

In the example explained in the last section, in particular in the body of function template `twice`, we apply the operation + to two parameters `x` of type `T`, and we allow only those `T` that satisfy the concept `Addable` constraints to represent `x`. In this case, the constraint will be `operator+`. This constraint represents an *implicit* parameter, or in other words, an abstract operation. This means we cannot call the `operator+` directly in the evaluation. Due to this, a very important question arises: How does a type satisfy a concept? There are a couple of debates about how this question should be answered; I will concentrate on how we choose to handle it in our interpreter, Liz.

Imagine you had the following:

```
struct Complex { ... };

Complex operator+(Complex x, Complex y) { ... }
```

and you make the call,

```
twice(Complex(2, 3), Complex(2, 3));
```

then Liz will follow by:

1. Deducing values for explicit parameters

   In this particular case we only have one explicit parameter, `T`, and Liz will deduce its value is equal to `Complex`

2. Deducing values for implicit parameters

   An implicit parameter will be any signature declaration we have in the concept, in this case `operator+` will be the only one. Liz uses the explicit parameters from the first step to substitute in the concept definition and deduce that the value for `operator+` is the `operator+` from the `Complex` struct.

After deducing all values for the parameters, Liz makes the appropriate substitutions, and the type, implicitly, satisfy the requirements imposed by the concept. We choose this kind of conformance, the *implicit* rather than *explicit* conformance, mainly because if programmers have to always specifically say what type satisfy some concept, then it will become very difficult to scale and essentially, a burden. We note that there would be cases where programmers have to explicitly say something in case of ambiguities. However, our hope is that a good system should make ambiguities rare.

Evaluator

After it performs the type checks, the elaborator will generate some code that will eventually be fed into the evaluator. This intermediate language is much simpler than Liz's

language, and unlike the language explained in the Parser section, it will be composed of the values and expressions shown in Fig. 2:

Value:	*Expression:*
Bool	*SymbolicReference(identifier, type)*
Int	*Bind(identifier, type)*
Double	*Write(address, value)*
String	*Read(expression)*
Function	*If(condition, expression)*
Type	*While(expression, condition)*
Concept	*Block(expression sequence)*
Axiom	*Call(expression, expression sequence)*
	Break
	Return(expression)

Fig. 2. Liz's Intermediate Language

The grammar of the intermediate language produced by the elaborator is substantially smaller than the grammar for C++. This makes the evaluation process, and actual code, much simpler.

The evaluator uses a Visitor Design Pattern (Gamma *et al.*, 1995), in which a visitor class is created and it implements all of the appropriate specializations of a virtual function. This is done in order to allow the adding of new virtual functions to a family of classes without modifying the classes themselves.

Control flow

Our evaluator can make a recursive call to itself in order to evaluate parts of a statement. If a control operator, like a return-statement, is encountered, we ran into an issue: to which point in the recursive call chain should control be transferred? It is not necessarily correct, and in fact almost always wrong, for the evaluator to just return to its caller. Rather, it must return to the point where the evaluator started evaluating the most recent function call.

To understand how we dealt with this issue let us take a look at a simple example. Suppose you had the following function,

```
int factorial(int n) {
    if (n < 2)
        return 1;
    else return n * f(n-1);
}
```

and you called the self evaluated expression,

`factorial(3);`

the evaluator will follow by,

1. Binding 3 to `n`
2. Evaluating `n < 2`, and if holds evaluating `return 1`
3. If not, evaluating `return n * f(n-1)`

In this example, the second step would not hold, as 3 is not less than 2, and the expression `n * f(n-1)` will be evaluated. A recursive call to `factorial(2)` will be made and the three steps stated above will be executed one more time. As `2 < 2` is false, there will be a third recursive call to factorial(1), in which `1 < 2` will hold, and the statement `return 1` will be executed; but where exactly does the machine return 1 to? As we explained earlier, it is not to the last caller, but to the most recent dynamic scope set, in this case to the call to `factorial(1)`. We achieve this with *exceptions*, which in C++ are treated as non-local goto's.

In liz, we have two additional non-local goto types, which are not yet completely implemented:

1. `goto`, which at this moment is not handled

2. `break`, which can be used in `while`, `switch`, and `do` statements. At this moment, the evaluator does not take into account if the `break` is used inside of the appropriate expressions or not.

CHAPTER III

USABILITY

We are now aware of the parts that conform Liz, and its internal workings. However, you may be wondering: how do we make use of concepts and axioms in Liz? In this chapter, we show how to accurately express our ideas in Liz using these features. The examples discussed in this chapter are based on algorithms from the Standard Template Library, as these are widely used and understood.

Once again: The problem

Let us take a simple algorithm, `copy`, and analyze it. The abstract idea of this algorithm is to essentially copy a sequence of elements into another sequence of elements. However, if we take the STL version of the algorithm,

```
template<class InputIterator, class OutputIterator>
    OutputIterator copy ( InputIterator first, InputIterator last,
                          OutputIterator result )
{
    while (first != last)
        *result++ = *first++;
    return result;
}
```

although it seems to express an abstract idea of what `copy` is supposed to do, it actually leaves out several important assumptions. Let us see a simple example of how this can lead to problems. Using the above code, if a programmer writes,

```
    vector<int> v(10);
    copy(7,17,v.begin());
```

where a value of type `int` has been used as an *iterator*, a huge and very hard to understand message will result. Although it seems like a perfectly honest and simple mistake, the programmer would get bombarded by many error lines seemingly referring to nothing in particular. If we were able to express our ideas clearly in the actual code, then we would be able to avoid these type of errors. Liz can help us achieve this.

Finding solutions

Let's analyze a simple example, the `find` algorithm of the STL. The abstract idea of this algorithm is, as it names indicates, to find the first position of a specific value in a list of elements. For this algorithm to function we need a sequence of elements and the value we are trying to find in that sequence. Let's take iterators to the sequence, and call the iterator to the beginning of the sequence `first` and the one-past-the-end iterator of the sequence `last`. We will refer to the value that we are trying to find as `value`. What properties should these iterators posses? What assumptions should we make? In order to match the value given with one of the elements of the sequence, we need to read from the sequence. Therefore, iterators `first` and `last` should be `Readable`, which according to the definition in the *Elements of Programming*, refers to the ability to obtain the value of an object denoted by another. Let's take a look at the STL version of this algorithm,

```
template<class InputIterator, class T>
  InputIterator find ( InputIterator first, InputIterator last, const T& value )
  {
    while (first != last && *first != value)
      ++first;
    return first;
  }
```

It is straight forward and easy to follow, but some requirements are missing. The fact that `first` and `last` should be `Readable` and `Iterators` is not expressed as part of the

algorithm. As we saw in the previous section this can lead to problems.

Let's analyze how we will write `Iterator` and `Readable` as concepts in Liz. The `Iterator` concept p. 91 will be written as,

```
concept Iterator(Regular T) {
    Integer DistanceType(Iterator);
    T successor(T);
}
```

where template `successor`, as its name specifies, is more or less like the "++" (or "+1") operation in C++, which lets you transition to the next item in a sequence. The type function `DistanceType` returns an integer type large enough to measure any number of applications of `successor`. A reduced version of concept `Integer` can be written as,

```
concept Integer(Regular I) {
    I successor(I);
}
```

which specifies that an integer type must provide the `successor` capability. According to the *Elements of Programming* a type T is *readable* if a unary function `source` defined on it returns an object of type ValueType(T). If we mirror what is in the book, the concept `Readable` p. 90 will be written in Liz as,

```
concept Readable(Regular T) {
    Regular ValueType(Readable);
    ValueType(T) source(T);
}
```

but how do we type check that concept? how do we find the definition of `ValueType`, and if we do, what should it look like? We do not have answers for these questions, but we have implemented a different system to deal with this issue. Let us take a look at the English specification: A type T is *readable* if there exists a function `source` taking an iterator of

the type T and returning a value of type depending on T, which is called `ValueType` in the book. If we translate this into code it would look like this,

```
concept Readable(Regular T) {
    exists(Regular V) {
        V source(T);
        ValueType = V;
    }
}
```

which is exactly what we want to say. In this case, we would not need to type check a function called `ValueType` because it will be implicitly defined by the compiler through the return type of the function `source`. This solution is based on Skolemization. Currently, this is implemented in the parser, but not yet in the type checker. However, we strongly believe it will work as predicted.

The unary function `source` is defined for all pointer types and returns a corresponding constant reference. We can write `source` in terms of function `deref`, which is built-in. The only difference between them is that `deref` is defined only for pointer types to nonconstant objects and it returns a nonconstant reference. We can define `source` as,

```
template<typename T> requires Mutable(pointer(T))
    const T& source (pointer(T) p) {
        return deref(p);
}
```

where *mutability* refers to the combination of *readability* and *writability* in a consistent way. We need `pointer(T)` to be `Mutable` because this will allow the replacement of `source` with `deref`, as we have done in the body of the template, without affecting a program's meaning. Now that we have defined the concepts and template functions used by our algorithm, we can write our version of `find` in Liz,

```
template<typename InputIterator>
    requires(Readable(InputIterator) && Iterator(InputIterator))
InputIterator find(InputIterator first, InputIterator last,
                   const ValueType(InputIterator)& value)
{
    //Precondition: readable_bounded_range(first,last)
    while (first != last && source(first) != value)
        first = successor(first);
    return first;
}
```

The above code, to the contrary of the STL algorithm, clearly specifies all requirements. The correspondence between the abstract idea of `find` and our computer program is obvious.

Wrapping it up

Let's go back to our first example, `copy`: What do we need to write such an algorithm with *concepts* and *axioms*? First of all, we require the sequence that we want to copy, and the sequence that we want to copy it to. Let's take iterators to these sequences, and we will call the iterator pointing to the first element in the sequence we want to copy `first`, and the iterator pointing to the last element in the sequence we want to copy `last`. We will call the iterator pointing to first slot of the result sequence `result`. Now, what kind of iterators should `first`, `last`, and `result` be? What properties should they follow? Remember that we are trying to read from one sequence and write to another one. Following that, iterators `first` and `last` should be `Readable`. In the same manner, iterator `result` should be `Writable`, which would allow its value to be modified.

The STL algorithm for `copy`, shown in the first section of this chapter, is missing a few requirements. The fact that `first` and `last` are `Readable` and `result` is Writable is not part of the algorithm; along with the fact than `first`, `last`, and `result` are *Iterators*.

Writability allows the value of some iterators to be modified. According to the definition of the *Elements of Programming*, a type is *writable* if a unary procedure `sink` is defined on it. Based on our previous explanation of the `Readable` concept, the `Writable` concept will be expressed in Liz as,

```
concept Writable(Regular T) {
    exists(Regular V) {
        V sink(T);
        ValueType = V;
    }
}
```

where function `sink` can be defined as follow,

```
template<typename T>
    T& sink(pointer(T) p) {
         return deref(p);
}
```

We have already discussed the requirements we need to have in order to express our abstract idea of `copy` correctly. We can now discuss what we would need in the body of our algorithm. As we have said previously, we want to copy a sequence of elements into another sequence of elements. For that we need to be able to move through the sequences, which we can do with `successor`; to read from the input sequence, which we can do with `source`; and to write to the output sequence, which we can do with `sink`. Finally, we can write our Liz version of `copy` p. 151-152,

```
template< Iterator InputIterator, Iterator OutputIterator >
    requires(Readable(InputIteator) && Writable(OutputIterator) &&
             ValueType(InputIterator) == ValueType(OutputIterator))
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result)
    requires (not_overlapped_forward(first, last, result, result + (last-first)))
{
   while (first != last) {
        sink(pointer(result)) = source(pointer(first));
        pointer(first) = successor(pointer(first));
        pointer(result) = successor(pointer(result));
    }
    return result;
}
```

where function `not_overlapped_forward` ensures that if the input and output ranges overlap, no input iterator is read after an aliased output iterator is written. Going back to our original problem, if we try to write the faulty program with the above code,

```
vector<int> v(10);
copy(7,17,v.begin());
```

you would get a clear error message stating that the arguments for `copy` do not match the definition. As we had said before, *concepts* and *axioms* would allow the exact translation of an abstract idea into code. More importantly, this exact translation would allow the compiler to *see* simple errors, like the above, and catch them on time. Saving a lot of time and effort to the programmer.

CHAPTER IV

CONCLUSION

Throughout this thesis we have discussed an important issue, proposed a solution, and implemented a tool to support it: our interpreter Liz. In Chapter I, we got an overview of the problem and its importance; in Chapter II, we got to know the internal workings of Liz; and finally, in Chapter III, we got to see examples of how to use Liz, and the benefits that we can get when using *concepts* and *axioms*.

The importance of these new features can be clearly observed when faced with an obscure messages after an unsuccessful linking. *Concepts* and *axioms* could save programmers a lot of time and effort by making those type of error disappear. Furthermore, this addition could make *templates* less intimidating. Overall, this project concretely shows, through Liz, the benefits that *axioms* and *concepts* could bring.

Currently, there is still much work to be done to Liz, and much more to be learned. However, I believe this project will continue to evolve and grow during the next couple of years.

REFERENCES

Czarnecki, Krzysztof, Eisenecker, Ulrich, & Czarnecki, Krzysztof. (2000) *Generative Programming: Methods, Tools, and Applications*. New York: Addison-Wesley Professional.

Dehnert, James C., & Stepanov, Alexander. (1998) Fundamentals of Generic Programming. In *Proceedings of the Dagstuhl Seminar on Generic Programming*. Lecture Notes in Computer Science, vol. 1766. London, UK: Springer-Verlag, pp. 1–11.

Dos Reis, Gabriel, & Stroustrup, Bjarne. (2006) Specifying C++ Concepts. In *Proceedings of the 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York: ACM Press, pp. 295–308.

Dos Reis, Gabriel, Stroustrup, Bjarne, & Meredith, Alisdair. 2009 (June) *Axioms: Semantics Aspects of C++ Concepts*. Tech. rept. N2887=09-0077. ISO/IEC SC22/JTC1/WG21. `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2887.pdf`.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley Professional.

Gärtner, Bernd, & Veltkamp, Remco. (2007) Editorial: A decade of CGAL. *Comput. Geom. Theory Appl.*, **38**(1-2), 1–3.

Gibbons, Jeremy, & Jeuring, Johan (eds). (2003) *Generic Programming*. Berlin Heidelberg: Springer-Verlag.

Gregor, Douglas, Järvi, Jaakko, Siek, Jeremy, Stroustrup, Bjarne, Dos Reis, Gabriel, & Lumsdaine, Andrew. (2006) Concepts: Linguistic Support for Generic Programming in C++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications*. New York: ACM Press, pp. 291–310.

ISO. (2003) *International Standard ISO/IEC 14882. Programming Languages — C++*. 2nd edn. Geneva: ISO.

Musser, David A., & Stepanov, Alexander A. (1988) Generic Programming. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. Lecture Notes in Computer Science, vol. 358. London, UK: Springer, pp. 13–25.

Musser, David R., & Stepanov, Alexander. (1994) Algorithm-oriented Generic Libraries. *Software–Practice and Experience*, **24**(7), 623–642.

Siek, Jeremy G., & Lumsdaine, Andrew. (1998) The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*. London, UK: Springer-Verlag, pp. 59–70.

Stepanov, Alexander, & Lee, Meng. 1994 (May) *The Standard Template Library*. Tech. rept. N0482=94-0095. ISO/IEC SC22/JTC1/WG21.

Stepanov, Alexander, & McJones, Paul. (2009) *Elements of Programming*. Boston: Addison-Wesley.

Stroustrup, Bjarne. 2009a (June) *Simplifying the Use of Concepts*. Tech. rept. N2906. ISO/IEC SC22/JTC1/WG21.

Stroustrup, Bjarne. (2009b) The C++0x "Remove Concepts" Decision. *Dr. Dobb's Journal*. `http://www.ddj.com/cpp/218600111?pgno=1` Overload Journal, Vol 92. August 2009.

Stroustrup, Bjarne, & Dos Reis, Gabriel. 2003a (September) *Concepts — Design Choices for Template Argument Checking*. Tech. rept. N1522. ISO/IEC SC22/JTC1/WG21.

Stroustrup, Bjarne, & Dos Reis, Gabriel. 2003b (September) *Concepts — Syntax and Composition*. Tech. rept. N1536. ISO/IEC SC22/JTC1/WG21.

Veldhuizen, Todd L. (1998) C++ Templates as Partial Evaluation. *CoRR*, **cs.PL/9810010**.

Villoria, Carla, & Dos Reis, Gabriel. 2009 (October) *Implementing a Parser for Elements of Programming*. Tech. rept. TAMU-CSE-2009-10-01.

Wirth, Niklaus. (1977) What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? *Commun. ACM*, **20**(11), 822–823.

CONTACT INFORMATION

Name:	Carla Villoria
Professional Address:	c/o Dr. Gabriel Dos Reis Department of Computer Science and Engineering Parasol Laboratory, 410C HRBB Texas A&M University College Station, TX 77843
Email Address:	villoria.carla@gmail.com
Education:	B.S., Computer Science, Minor in Business Texas A&M University, May 2011 Undergraduate Research Scholar Phi Kappa Phi